AN ANALYSIS OF PARAMETER EVALUATION OF RECURSIVE PROCEDURES

Lawrence Snyder

Carnegie-Mellon University

Prepared for:

Air Force Office of Scientific Research
Advanced Research Projects Agency

April 1973

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD770 631

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 73 - 2003	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) AN ANALYSIS OF PARAMETER EVALUATION FOR RECURSIVE PROCEDURES		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Lawrence Snyder		8. CONTRACT OR GRANT NUMBER(s) F44620-70-C-0107
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Department of Computer Science Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D AO 827
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE April 1973
		13. NUMBER OF PAGES 110
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research (NM) 1400 Wilson Blvd Arlington, Virginia 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Let P_{R_X} be the class of recursive program schemata using method x as a parameter binding mechanism for procedure arguments. Methods of call by value (v), call by copy (c), call by reference (r), call by name (j), and "normal" evaluation (n) are studied. Call by name is "stronger" than the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Item 20. Abstract (Continued)

others ($P_{R_x} < P_{R_j}$, x = V,c,r,n) not because evaluation is postponed, but because it is repeated. Markers and global varialbes augmentation is studied and one result is $P_{R_{jm}}$ is"universal". $P_{R_j} < P_{R_{jm}}$ is conjectured and discussed. Relationships with data structures and fixed point theory are discussed as well as some decision problems.

ib.

AN ANALYSIS OF PARAMETER EVALUATION FOR RECURSIVE PROCEDURES

by

Lawrence Snyder

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213
April, 1973

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) monitored by the Air Force Office of Scientific Research, and in part by The Shell Oil Companies Foundation. This document has been approved for public release and sale; its distribution is unlimited.

Abstract

A class of recursive program schemata P_R is defined abstracting ALGOL-like procedures. Four generic types of parameter evaluation are considered: call by value, P_{Rv}, call by copy, P_{Rc}, call by reference, P_{Rr}, and call by name, P_{Rj}. Two other types of theoretical interest are also considered: "normal evaluation", P_{Rn}, a non-side effect, evaluation postponement mechansim and call by quote, P_{Rq}, an extension of call by name with all assignments to formals unevaluated. For evaluation method x, augmentation with global variables, P_{Rxg}, and augmentation with a finite number of markers, P_{RxM} are also considered. The results include,

$$P_{Rv} \equiv P_{Rc} \equiv P_{Rr} \equiv P_{Rn} \equiv P_{Rxg} \equiv P_{RxM} < P_{Rj} \equiv P_{Rjg} \leq P_{RjM} \equiv P_{Rq}$$

where x = v, c, r, and n. The results are correlated with the notion of fixed point computation and although call by value, copy and reference do not, in general, compute the least fixed point, for any functional an equivalent one may be found for which these three do compute the least fixed point.

That call by name is stronger than the other generic evaluation mechanisms derives not from postponing evaluation, but from repeated evaluation, i.e. side effects. The last two terms in the relation list are "universal". Additionally, its shown that if $P_{Rj} \equiv P_{RjM}$ and if $P_{Rj} \equiv P_{Rq}$, then they cannot be constructively equivalent.

An abstract model of parameter evaluation is developed and the constituent components of parameter evaluation are isolated. These help to characterize what impact choices of parameter evaluation have on the language in general. This understanding motivates the class P_{Rq} and binds our analysis with that of the lambda calculus models. The question of recognizing when two parameter evaluation mechanisms coincide is studied, i.e. when two mechanisms always compute the same result for the same schema. The general problem is not partially decidable for seperable evaluation mechanisms. For free recursive schemata employing the generic mechanisms, the coincidence question is decidable.

A discussion of the utility of such analysis and some thoughts on future directions for research are included.

Contents

Acknowledgement.

To my Professor and thesis committee chairman, A. N. Habermann, I wish to express my deepest appreciation for his patient understanding, his assistance and his counsel thoughout my studies at CMU. I wish also to thank Professors J. Grason, D. Loveland, A. Newell and D. Parnas for serving as my thesis committee. To my colleagues, the "brownie plate" group, I wish the best of luck.

1. Introduction.

1.1 Motivation and Overview.

"Parameter evaluation," "argument substitution," and "formal - actual binding" are some of the names which have been used to describe the object under investigation in this thesis. The activity of parameter substitution has received considerable attention in many contexts, the ***lambda*** calculus being one example. Our motivation for studying this phenomenon is to understand the ***semantics*** of parameter evaluation in the context of programming languages. To clarify our point of view, consider these observations.

In ALGOL there are two methods of parameter evaluation defined for the language: call by value and call by name*. Call by value is a natural, uncomplicated and quite efficient method of evaluating parameters. It is also most familiar to us. In contrast, call by name is subtle, generally inefficient and extremely complex to implement. But that's not all. The presence of call by name among the repertoire of facilities in the language pervades the entire implementation of an ALGOL compiler. Many semantic issues, which would be easy to implement without call by name, have added complexity. For example, the "run-time" stack discipline would be far simpler were it not

---

* The term call by name has been applied to quite a few different parameter passing devices in the literature. Our understanding of this term is consistent with that given by Ekman and Froberg [5]. See also 2.1.5 for a definition.

for call by name. Optimizations such as common subexpression elimination would be considerably easier since the "order of evaluation" rules could be by-passed more easily. These are just two examples; a compiler writer could, no doubt, find a dozen.

Since ALGOL already has call by value, the question which one asks is "Does call by name provide any 'extra' facility in return for the added expense and complexity?" Of course, the question must be suitably formalized (and will be) to be answered, but one's intuitive feeling is to give an unqualified "yes". It is, indeed, powerful, convenient and flexible. Furthermore, one feels that the apparent advantage derives from the evaluation postponement which may avoid initiating potentially divergent computations. Call by name will be shown to be "more powerful" than call by value, but not because it postpones evaluation!

Call by name and call by value are not, however, the only methods by which functional parameters are bound to their formal representatives. Hence, our task will be more general than suggested above. We shall make a comparative analysis of several well known parameter evaluation mechanisms. In Chapters 2 and 3 we will study five generic parameter passing mechanisms and the results will provide one measure of how substantive the apparent differences really are among such mechanisms.

The comparative analysis will also suggest which components of a parameter binding mechanism are fundamental. This insight will then be used

in Chapter 4 to formulate an abstract model of the parameter binding activity as it is used in contemporary programming languages. The model will provide a convenient method of specifying the salient features of a particular mechanism. Such specification will expose the similarities and differences in the mechanisms discussed before. Even the consequenses of some simple changes in definitions will become predictable. More importantly, it will suggest different ways in which parameter evaluation might be generalized. A generalization will be chosen and the consequences will be explored.

Finally, this model of parameter evaluation, which enables its decomposition into constituent parts, will be used to study questions of interest to a compiler writer. In particular, we recognize that parameter evaluation provides several semantic facilities, for which the compiler writers must generate code. In Chapter 5 we address the question of recognizing when these particular facilities are used. The assumptions made by compiler writers are made explicit and in this context we consider decidability questions for semantic constituents of parameter binding. To the extent that this is successful, the compiler writer can perform optimizations.

Chapter 6 discusses the overall utility of this type of analysis. Directions for further research will be suggested.

1.2. Background.

This thesis has its origins in two areas: the art of compiler writing and language design and the theory of program schemata. The former suggested the questions; the latter provided some answers. The literature on compiler writing and language design is plentiful enough, but little of it is relevant since it tends to deal with all features of a particular language rather than a particular feature of many languages as we intend to do. Hence, each parameter evaluation mechanism is specifically referenced when it is introduced. The reader may consult [13] for a description of several of them, however. For compiler related issues, our vocabulary is basically consistent with that used in Randell and Russell's book on ALGOL implementation [19].

By comparison with programming languages the literature on program schemata is less plentiful (although it is still substantial.) Since we draw on work by several investigators, we now present a brief introduction to the relevant literature to enable the reader to understand the context in which we work.

Although the interest in the last five years has become quite intense, the earliest work is generally attributed to I. Ianov [11] in 1958 (see Rutledge [21] for a translation and generalization.) Ianov's model was that of a simple flowchart language with one location. Many questions were decidable since the model was equivalent to a finite automaton. The Russians

followed the path of adding limited generalizations and studying equivalence preserving transformations. Ershov [6] provides a good review. Meanwhile, Luckham and Park showed in 1964 that the equivalence problem for flowchart schemata (with two or more locations) was unsolvable. Subsequent results by Paterson [17] and Luckham, Park and Paterson [14] showed that weaker notions of equivalence are also undecidable.

The unsolvability results have led investigators to study program schemata with restrictions. Among these are loop-free schemata, free schemata (any path is an execution path) and liberal schemata (no expression recomputation) [17], monadic schemata (one place functions and predicates) and independent location, monadic schemata [14]. We will also find that unsolvability prevents us from fully exploring all of the questions of interest to us and so we will adopt the free schemata restriction in our later work.

Recursive schemata have also received considerable study. These come in several syntactic varieties, the most frequently used having been introduced by McCarthy [16]. Strong [22] has studied the translatability of these recursive equations into flowchart schemata. Garland and Luckham have considered a restricted version of this problem: the translatability of monadic recursion schemata.

Paterson and Hewitt [18] appear to have initiated a type of schematic study which they termed *comparative schematology*. This study can be defined as the comparison of classes of functionals defined by different

classes of schemata. Typically, the two classes will be defined so that one has an extra semantic facility available. Then any differences in the classes of functionals defined are attributed to that construct. Other studies in comparative schematology have been carried out by Hewitt [9], Constable and Gries [4] (upon which much of this work is based), Brown [1], and Brown, Gries and Szymanski [2].

The techniques of comparative schematology will be employed in what follows. In addition, we will want to use results consistent with those of other authors. Thus, we present in the next section some definitions and a few pertainent results.

1.3. Definitions and Earlier Results.

The vehicle for our investigation of parameter evaluation mechanisms is a class of recursive program schemata. Our definition is essentially that one introduced by Constable and Gries [4]. Other models of recursion might have been used, but this seems a more reasonable model of languages such as ALGOL and PL/1. We will add the apparently trivial generalization that auxiliary functions (i.e. user defined functions) may appear as actual parameters to auxiliary functions. As is shown later, this simple generalization can have quite important consequences for certain parameter evaluation strategies, without seriously perturbing this simple model.

This definition is to be a "complete" specification of recursive program schemata without any specification as to the way in which parameters are to be evaluated. We will define separate classes for each kind of evaluation by augmenting this class with the required semantics.

Let V, F, P, L and C be disjoint enumerable sets of symbols called variable names, basic function names, basic predicate names, label identifiers and auxiliary function names, respectively.

Definition 1.3.1: Define the productions* of the grammar G with sentence symbol <rec program> to be:

	<rec program>	::=	<program>{<function def>}
	<program>	::=	<V - list> : <body>
(*)	<function def>	::=	<aux function>[<V - list>] : <body>
	<body>	::=	<S - list>; [<label>:] **halt**(<variable>)
	<S - list>	::=	[<label>:]<S> {; [<label>:]<S>}
	<V - list>	::=	(<variable> {, <variable>})
	<S>	::=	"empty"
			| <variable> ← <term>
			| **if** <predicate>[<V - list>]
			then [<label>:]<S> **else** [<label>:]<S>
			| **halt**(<variable>) | **goto** <label>
			| **begin** <S - list> **end**

---

* Here we use extended BNF notation where {t} and [t] mean "zero or more occurrences of t" and "zero or one occurrence of t," respectively.

<term> ::= <variable>

| <basic function>[<V - list>]

(**) | <aux function>[<T - list>]

<T - list> ::= (<term> {,<term>})

where:

<variable> ∈ V

<basic function> ∈ F

<predicate> ∈ P

<label> ∈ L

<aux function> ∈ C

The grammar G will define the admissable syntactic form of our schemata. The notation Rf, Rp and RG will be used to denote rank (number of parameters) of elements f ∈ F, p ∈ P and G ∈ C, respectively.

Definition 1.3.2: A schema S in the class B of ***basic recursive program schemata*** is a terminal string in the language L(G) such that the following semantic restrictions apply:

(i) The variables in <V - list> preceding <body> for both <program> and <function def> must all be different elements of V.

(ii) The <aux function> names for any <function def> must all be different elements of C and they must all be defined.

(iii) The significance of <label>s and <variable>s of any <body> is restricted in scope to that <body> and

(a) all statement <label>s within a <body> must be different

elements of L and each L in **goto** L must be defined in that <body>,

(b) all <variable>s used in a <body> which are not in the <V - list> preceding that <body> are called locals and are initialized to undefined (Ω) prior to executing the <body>.

(iv) The arguments to basic function names and predicate names are evaluated sequentially, left to right.

(v) The result of any <body> is the value of the argument variable to the first **halt** statement encountered.

Our conventions for symbols throughout the thesis will be as follows:

variables names	$V = \{u,v,w,x,y,z,u_1,v_1,w_1, \ldots\}$
basic function names	$F = \{f,f_1,f_2, \ldots\}$
basic predicate names	$P = \{p,p_1,p_2, \ldots\}$
label identifiers	$L = \{L,L_1,L_2, \ldots\}$
auxiliary function names	$G = \{G,G_1,G_2, \ldots\}$

In later constructions we will use names not appearing in the above lists but which have been chosen for their special semantic content. The type of objects which the symbols denote should be clear from context, however.

Example:

(x): $z \leftarrow G(x)$; **halt**(z)

G(y): **if** $p_1(y)$ **then** $w \leftarrow f_1$;

else begin

$z \leftarrow G(f_3(y))$;

$w \leftarrow f_2(y,z)$ **end**

halt(w)

A sample schema, S, in the class B of basic recursive program schemata. One auxiliary function (G) is defined, three basic function names (f_1,f_2,f_3) and the basic predicate name (p_1) have been used.

As an example of adding a parameter evaluation mechanism to the basic class, we give the definition of the class P_R of recursive program schemata presented by Constable and Gries.

Definition 1.3.3: [4] A schema S in the class P_R of ***recursive program schemata*** (with call by value) is a terminal string in the grammar formed by replacing the production labeled (**) in the grammar G given above with

| <aux function>[<V - list>].

Actual parameter variables (the only kind of actuals in this class) have their values assigned to formal parameters prior to execution of the function and the formals are then treated just as locals.

The schema cannot be meaningful until a domain, D, has been specified. The domain is the class of objects which are manipulated by the schema, e.g. D = {0,1,2, . . .} or D = {strings over a finite alphabet}. Given D, the (total) basic functions and (total) basic predicates may be chosen from the classes $\mathcal{F}(D) = \{f_n : D^{Rf} \rightarrow D\}$ and $\mathcal{P}(D) = \{p_n : D^{Rp} \rightarrow \{true,false\}\}$, respectively. Finally, the

input variables are chosen from the domain and the schema may be "evaluated" with this interpretation. The way in which the evaluation takes place should be clear, since the constructs have their obvious meaning consistent with the comments in the preceding paragraph.

Example: Two interpretations for the schema S in the above example.

(1)		(2)	
	$D_1 \equiv \{0,1,2, \ldots\}$		$D_2 \equiv$ {LISP lists}
	$f_1 \equiv 1$		$f_1 \equiv$ NIL
	$f_2 \equiv \lambda x,y[x * y]$		$f_2 \equiv \lambda x,y[\text{cons}(\text{car}(x),y)]$
	$f_3 \equiv \lambda x[x - 1]$		$f_3 \equiv \lambda x[\text{cdr}(x)]$
	$p_1 \equiv \lambda x[x = 0]$		$p_1 \equiv \lambda x[x = \text{NIL}]$

The schema $S[f_1,f_2,f_3,p_1](x)$ instantiated according to (1) defines x! over D_1 and instantiated according to (2) defines *reverse*(x) over D_2.

Evidently, a schema S with r basic function names, s basic predicate names, and t input variables and a domain D define a mapping

$$S: \mathcal{F}(D)^r \times \mathcal{P}(D)^s \rightarrow [D^t \rightarrow D]$$

of basic functions and predicates into the class of functions from t-tuples of domain values to domain values. These mappings are called ***functionals*** over D. The set of all functionals over D computable by schemata in the class B is denoted FUNC(B,D). The use of functionals allows a comparison of schemata classes with differing data and control structures. Two other classes of interest will be defined below. Our comparison of classes of schemata requires the following definition.

Definition 1.3.4: [4] Two schemata S_1 and S_2 with r basic function names, $f_1, \ldots, f_r$, s basic predicate names $p_1, \ldots, p_s$ and t input variable names $v_1, \ldots, v_t$ are ***equivalent over*** D if and only if

$$S_1[f_1, \ldots, f_r, p_1, \ldots, p_s](v_1, \ldots, v_t)$$
$$= S_2[f_1, \ldots, f_r, p_1, \ldots, p_s](v_1, \ldots, v_t)$$

for all values of v_i in D. Two schemata are ***equivalent*** if and only if they are equivalent for all D. Thus, they are equivalent if they compute the same functionals over all D.

Definition 1.3.5: [4] Two classes of schemata C_1 and C_2 are related by $C_1 \leq C_2$ if for every schema $S_1 \in C_1$ there exists a schema $S_2 \in C_2$ such that S_1 is equivalent to S_2. $C_1 \equiv C_2$ if and only if $C_1 \leq C_2$ and $C_2 \leq C_1$, i.e. $\text{FUNC}(C_1,D) \equiv \text{FUNC}(C_2,D)$. $C_1 < C_2$ if and only if $C_1 \leq C_2$, and not $C_2 \leq C_1$.

In order that we may integrate our results with those which have preceded ours, we now present definitions for two nonrecursive classes. The class, P, of flowchart schemata has been extensively studied (see 1.2). Array schemata were introduced by Constable and Gries [4]. It will be convenient to refer to both classes in our subsequent work.

Definition 1.3.6: The schema S in the class P of ***simple flowchart schemata*** is a terminal string in the grammar formed by deleting the two productions marked (*) and (**) from grammar G.

Definition 1.3.7: [4] The class P_A of ***program schemata augmented with arrays***

is the class P with the additional productions:

<S> ::= v ← 0 | v ← w + 1

<term> ::= A[v]

and a one dimensional array A of simple variables A_0, A_1, ... To allow subscripting, a subscript set $N = \{0,1,2, \ldots\}$ is provided which is disjoint from the domain D. Any simple (or array) variable may take values in D U N. The term A[v] refers to A_v if the value of v ∈ N and A_0 otherwise. If the value of some variable v is in N and v is used as an argument to a basic function, basic predicate or to a **halt** statement the value Ω is used instead. The remainder of the semantics should be obvious and the reader may consult [4] for further details and a discussion of how reasonable a model this is of array languages.

Assertion 1.3.8: [4] $P < P_R < P_A$.

Assertion 1.3.9: [4] P_A is "universal" for total interpretations.

The notion of universal means intuitively that the class computes all the functionals computable in an effective way. In [4] Constable and Gries show several classes which are equivalent to P_A representing several kinds of semantic facilities. Brown [1] and Brown, Gries and Szymanski [2] have found others. An interesting feature is that although there are a number of universal classes, P_A cannot be constructively equivalent to most of those investigated thus far. We shall have more to say about this in Chapters 3 and 4. The interested reader should consult [4] for a discussion of

universality.

Assertion 1.3.10: [14] For a schema $S \in P$, it is not partially decidable whether S diverges under all interpretations.

This completes the definitions and results required to properly present our results and correlate them with that which has gone on before. In what follows, we adopt the usual procedure of "proving" our results by presenting a construction to justify our claims. The constructions are generally sufficiently simple so that we may avoid the tedium of proofs by induction on state transitions and similar proof methods.

2. The Weaker Parameter Evaluation Mechanisms.

2.1. Definitions.

In this section we employ the class of basic recursive program schemata (see 1.3) to define several classes of schemata which employ differing parameter passing mechansims. One can consider quite a few different ways in which formals and actuals could be made to correspond, but our intention is to study those mechanisms which are actually in use or which contribute to our understanding of the subtleties of other mechanisms. Hence our attention will initially focus on five methods:

call by value (e.g. ALGOL)

call by copy (e.g. WATFOR)

call by reference (e.g. certain versions of FORTRAN)

"normal" evaluation (see below)

call by name (e.g. ALGOL)

When these have been fully explored, we will direct our study towards variations. First we discuss what facilities are provided by these generic mechanisms.

The formal notations available to us for specifying semantic notions either omit or obscure the detail which we wish to present in the following definitions of parameter evaluation. Therefore we are left to describe these concepts in English. To make this task easier, let us define ***formal parameter*** as a parameter in the formal specification of an auxiliary function. An ***actual***

parameter is a parameter which is used in a function call. In example 2.1.3, x is an actual and y is a formal. Note that within the function definition formals may, in turn, be used as actuals.

The five types of recursive program schemata which we will consider vary only in the semantics of the parameter passing and formal variable reference. These differences manifest themselves at three different points in the "interpreter":

(i) ***auxiliary function initiation*** - operations performed before execution of the <body> of the function begins.

(ii) ***interpretation of formal parameters*** - method by which the value of the formal, referenced in the text, is found.

(iii) ***auxiliary function termination*** - operations performed before execution resumes at the point of the call.

Thus the definitions given below will concentrate on the semantics at these three points. Regardless of the kind of parameter evaluation, auxiliary function initiation (i) includes the initialization of the local variables to undefined (Ω) and auxiliary function termination (iii) includes returning the value of the argument of the **halt** as value of the function. When auxiliary function termination only involves returning the function value, it is elided in the following definitions.

Definition 2.1.1: A schema in the class P_{RV} of ***recursive program schemata with call by value*** parameter evaluation is a basic recursive

program schema such that in the interpretation:

(i) auxiliary function initiation: the actual parameters are evaluated sequentially, left to right. The formals are then initialized to the value of their corresponding actuals.

(ii) interpretation of formal parameters: the same interpretation as that of local variables.

Definition 2.1.2: A schema in the class P_{Rc} of ***recursive program schemata with call by copy*** parameter evaluation is a basic recursive program schema such that in the interpretation:

(i) auxiliary function initiation: as in 2.1.1(i).

(ii) interpretation of formal parameters: as in 2.1.1(ii).

(iii) auxiliary function termination: the value of each formal parameter is copied into its corresponding actual parameter if and only if the actual is a simple variable. Copying proceeds sequentially, left to right.

Example 2.1.3:

(x): $w \leftarrow G(x)$; **halt**(x)

G(y): $y \leftarrow f_1(y)$; **halt**(y)

A sample schema S. The functionals defined are as follows: if $S \in P_{Rv}$ then $S[f_1](x) = x$ and if $S \in P_{Rc}$ then $S[f_1](x) = f_1(x)$.

Thus call by copy allows computations from a called function to be returned to the calling environment by way of parameters as well as the

normal value returning mechanism of the recursive function. Such a facility is often convenient but there is an alternative way of realizing a similar effect as we now see.

Next we introduce call by reference parameter evaluation. Call by reference is a natural method of passing parameters for computers since the reference to a value is merely its "memory address." Although memory addresses have not been introduced, they need not be since we can achieve the same effect by admitting variable names (elements of V) as values. Thus, if x is a formal in a schema with call by reference then its value will be a name, say y, out of V. We will call y the *reference* value of the formal x. If the value of y is a value from the domain, say d, then we say the *coerced* value of the formal x is d. If the value of y is a reference value, then the coerced value of x is, inductively, the coerced value of y.

Definition 2.1.4: A schema in the class P_{Rr} of *recursive program schemata with call by reference* parameter evaluation is a basic recursive program schema such that in the interpretation:

(i) auxiliary function initiation: if for any formal x the corresponding actual is a function call (basic or auxiliary) then a *surrogate* variable $w \in V$ is chosen which does not appear elsewhere in the schema.

The formals are assigned values, proceeding sequentially left to right, as follows. Let x be a formal, then assign to x:

(a) the *name* of the surrogate variable if the corresponding actual is a function call, (i.e. x has reference value w.) The function is evaluated and the value is assigned to the surrogate w.

(b) the *name* of the corresponding actual if the actual is a simple variable but not a formal.

(c) the *reference value* of the corresponding actual if it is itself a formal.

Note that in (c) the formal is assigned the reference value of its correspondent and so, there can be at most one level of indirection, (i.e. no formal can have a reference value which is itself a reference value.) This is equivalent to an arbitrarily long chain of references (see 4.1) but it simplifies our constructions.

(ii) interpretation of formal parameters:

(a) if the formal x is not on the left hand side of an assignment then the value is the coerced value of x.

(b) if the formal x is on the left hand side of an assignment, then the assignment is to the reference value of x, (i.e. the coerced value of x is changed, not its reference value.)

Call by copy and call by reference are frequently thought to be two methods of realizing the same facility, namely, returning information by way of the parameters. The following example indicates that these are not

necessarily the identical for all schemata.

Example 2.1.5:

(x): $u \leftarrow G(x,x)$; **halt**(x)

G(y,z): $y \leftarrow f_1(y)$; $z \leftarrow f_1(z)$; **halt**(y)

A sample schema S such that the functionals computed by call by copy and call by reference are as follows: if $S \in P_{Rc}$, then $S[f_1](x) = f_1(x)$ and if $S \in P_{Rr}$, then $S[f_1](x) = f_1(f_1(x))$.

In Manna, Ness and Vuillemin [15] a parameter evaluation mechanism, the "normal" evaluation rule, is introduced. It is shown that call by value is strictly "weaker" than "normal" evaluation by showing that the former is not a "fixed point" rule and the latter is. These results are shown for recursive equations and there can be no possible side effects. Although their model is somewhat different from ours (see 2.3), the following should be a faithful rendering of the intent of their definition. The relationship of our results to theirs will be discussed later.

Definition 2.1.6: A schema in the class P_{Rn} of ***recursive program schemata with normal parameter evaluation*** is a basic recursive program schema such that in the interpretation:

(i) auxiliary function initiation: no evaluation is performed.

(ii) interpretation of formal parameters: when a formal parameter is encountered for the first time in this environment the corresponding actual is evaluated and the value assigned to this

formal. It used for this and all subsequent formal references in this environment as though it was a local.

Clearly, with P_{Rn} parameter evaluation may be postponed or even avoided entirely. This has no effect in the case where parameters are simple variables or basic functions with non-formal parameters. But if an actual is an auxiliary function, it may be partial and hence postponement appears to offer an advantage over, say, call by reference.

Example 2.1.7:

(x): $u \leftarrow G_1(x,G_2(x))$; **halt**$(u)$

$G_1(y,z)$: **halt**(y)

$G_2(w)$: $w \leftarrow G_2(w)$; **halt**(w)

A sampie schema S such that, if $S \in P_{Rn}$ then $S[](x)$ = undefined and if $S \in P_{Rn}$ then $S[](x) = x$.

Next we introduce the ALGOL call by name parameter evaluation mechanism. Although it postpones evaluation just as normal evaluation does, it differs in a very important way. Each time an occurrence of a formal is encountered, it requires a separate evaluation. Thus the side effects of one evaluation may influence the results of a subsequent evaluation.

Definition 2.1.8: A schema in the class P_{RJ} of *recursive program schemata with call by name* parameter evaluation* is a basic recursive program

---

* Call by name is sometimes referred to as "Jensen's device" after Jørn Jensen, hence the class name P_{Rj}.

schema such that in the interpretation:

(i) auxiliary function initiation: no evaluation is performed.

(ii) interpretation of formals:

(a) if the formal is not the left hand side of an assignment then the corresponding actual is evaluated in the environment of the call. The resulting value is the value of the formal for this evaluation of this occurrence only.

(b) if the formal is the left hand side of an assignment then if the corresponding actual is a simple non-formal variable then it receives the value of the assignment. If the actual is a function call, then the entire assignment statement is ignored. If the corresponding actual is itself a formal this process is applied recursively with this formal as the left hand side.

Clearly, in evaluating a formal the actual corresponding to it may be a formal in the calling environment and thus one must return to earlier and earlier calling environments as long as the corresponding actual continues to be itself a formal. Note that in 2.1.8(ii)b, there are some consequences to our decision to ignore the entire assignment statement if the actual is not a simple variable but the corresponding formal is used on the left hand side of an assignment statement. In particular, one might consider evaluating both sides of the assignment statement and then disregarding the values. This would allow more "side effects" but would it actually enlarge the class of

functionals defined? It does not, but we postpone the proof until section 3.1.

Example 2.1.9:

(x): $u \leftarrow G_1(x,G_2(x))$; **halt**$(u)$

$G_1(y_1,y_2)$: $w \leftarrow f_2(y_2,y_2,y_1)$; **halt**$(w)$

$G_2(w)$: $w \leftarrow f_1(w)$; **halt**(w)

A sample schema S such that, if $S \in P_{Rn}$ then $S[f_1,f_2](x) = f_2(f_1(x),f_1(x),x)$ and if $S \in P_{Rj}$ then $S[f_1,f_2](x) = f_2(f_1(x),f_1(f_1(x)),f_1(f_1(x)))$.

To emphasize the differences between the parameter evaluation mechanisms just defined, we present a schema from L(G) together with the results of interpreting it as an element of each of the five different classes.

Example 2.1.10:

(u): $v \leftarrow G(u,u,f_1(u))$; **halt**$(u)$

$G(x,y,z)$: $x \leftarrow f_2(x)$;

$y \leftarrow f_3(y)$;

$y \leftarrow f_4(y,z)$;

halt(x)

(Note that the value of the schema is the actual parameter u.)

Class	*Schema Result*
P_{Rv}	u
P_{Rc}	$f_4(f_3(u),f_1(u))$
P_{Rr}	$f_4(f_3(f_2(u)),f_1(u))$
P_{Rn}	u

P_{Rj} $\qquad\qquad$ $f_4(f_3(f_2(u)),f_1(f_3(f_2(u))))$

In 1.3 we commented that allowing function as actuals would not change the results given earlier. This is immediate when one observes that for any schema $S \in P_{Rv}$ an equivalent schema in P_R can be constructed by merely expanding each call of the form

$$u \leftarrow G(\langle term\rangle_1,\langle term\rangle_2,\ldots,\langle term\rangle_{RG})$$

to be

begin $w_1 \leftarrow \langle term\rangle_1$;

$w_2 \leftarrow \langle term\rangle_2$;

. . . ;

$w_{RG} \leftarrow \langle term\rangle_{RG}$;

$u \leftarrow G(w_1,w_2,\ldots,w_{RG})$ **end**

where each w_i is a new variable. Thus our class of recursive schemata with call by value is equivalent to the one introduced by Constable and Gries. We refer to this technique as *coercing actuals*.

2.2. The Classes P_{Rc}, P_{Rr} and P_{Rn}

The examples of the last section illustrated that there is considerable diversity among the various parameter evaluation mechanisms. The differences are reflected in the convenience with which one writes programs, efficency of execution, the strategy of implementation of recursive execution and the ease with which one verifies that a program is correct. Our

interest in this section is in whether or not the differences are reflected in "what can be computed." We might expect that the facility of returning parametric values and the facility of evaluation postponement would both provide a computational advantage. We shall discover, however, that call by value, call by copy, call by reference and, surprisingly, normal evaluation are all equivalent in the sense that their respective functional classes are equivalent. In addition, we shall show that global variables and markers do not enhance the computing "power" of these classes of functionals.

Theorem 2.2.1: $P_{Rv} \equiv P_{Rc}$.

Proof: Using the construction described at the end of the previous section, all actuals of a schema $S \in P_{Rv}$ may be shielded from updating. The resulting schema may then be interpreted in P_{Rc}, yielding $P_{Rv} \leq P_{Rc}$. To see that $P_{Rc} \leq P_{Rv}$ all that is required is to show that the actuals may be updated. Clearly, if all actuals are functions, they are already treated properly. For each auxiliary function call containing any simple variables

$$v \leftarrow G(v_1, v_2, \ldots, v_{RG})$$

substitute

begin $w_1 \leftarrow G_1(v_1, \ldots, v_{RG})$;

. . . ;

$w_{RG} \leftarrow G_{RG}(v_1, \ldots, v_{RG})$;

$v \leftarrow G(v_1, \ldots, v_{RG})$;

$v_1 \leftarrow w_1$;

. . . ;

$v_{RG} \leftarrow w_{RG}$ **end;**

where:

w_i are new simple variables

G_i are new functions identical to G except that each **halt** statement has the i^{th} formal as argument.

Note that the case where the function is an actual (rather than a right hand side of an assignment) is handled by applying the procedure recursively. q.e.d.

Call by copy and call by reference are frequently thought to be two different ways of achieving the same effect. However, the previous examples show that there is a schema on which these two mechanisms differ. Call by reference formals refer indirectly to the sole instance of the actual while call by copy formals use a duplicate of the actual and update it at function termination. These two mechanisms are identical as long as different formals correspond to different actuals, since the reference discipline guarentees access of the single most recently assigned value corresponding to the actual. In the example 2.1.10 there were two separate values (corresponding to the two instances of u) and hence the difference. Thus, we need only handle this case to establish equivalence.

Theorem 2.2.2: $P_{Rc} \equiv P_{Rr}$.

Proof: From the preceding remarks we need only consider the case of

multiple instances of a variable actual in a single call. Suppose that some call

$$\ldots\ G(v_1,\ldots,v_{RG})\ \ldots$$

occurs in a statement in schema S and that parameter positions i,j, . . .,k are the same simple variable v.

(i) $S \in P_{Rc}$. To construct an $S' \in P_{Rr}$ change each such call to

begin $w_i \leftarrow v$;

. . . ;

$w_k \leftarrow v$;

. . . $G(v_1,\ldots,w_i,w_j,\ldots,w_k,\ldots,v_{RG})$ . . .

$v_i \leftarrow w_k$ **end**

where the w's are new variables. Clearly, if G is an actual to a function with some of the same multiply occurring actuals, G will have to be evaluated, assigned to a new variable and that variable used as the actual.

(ii) $S \in P_{Rr}$. To construct an $S' \in P_{Rc}$ change each call to

. . . $G_{i,\ldots,k}(v_1,\ldots,v_{RG})$ . . .

where $G_{i,\ldots,k}$ is a new procedure modified so that any assignments to the formals $u_i,u_j,\ldots,u_k$ corresponding to v, say,

$u_j \leftarrow$ <term>

are modified to read

begin $u_j \leftarrow$ <term>

$u_i \leftarrow u_j$;

. . . ;

$u_k \leftarrow u_j$ **end**

If in the body of G these formals are also actuals to some auxiliary function call then use only one of the formals in all positions and follow the call with the required updating of the other formals. Obviously, this last requirement may introduce some new function calls with multiply occurring actuals. But, if k is the number of auxiliary functions in S and n is the maximum number of parameters to any of these then there can be at most $k(2^n-n)$ ways of having multiply occurring instances of a single variable and hence new functions introduced. This assures that there are only a finite number of different kinds of calls and the construction reduces this number by one with each application. Clearly, if there are multiple occurring instances of several variables in a call, then the above procedures may be applied iteratively, since each application removes multiple instances of a variable without introducing new ones. q.e.d.

In [2] global variables were introduced by modifying the definition of <function def> as given in 1.3. as follows:

<function def> ::=

<aux function>[<V -list>]: **global** <variable>{,<variable>};<body>

If variables $w_1,...,w_n$ follow **global** then they refer to variables in the main <body> with the same names and cannot occur in the formal parameter list.

The $w_1,\ldots,w_n$ are not initialized. A class P_{Rx} augmented with **global** variables is denoted by P_{Rxg}. It was shown in [2] that global variables do not enlarge the class of functionals definable by P_R. This is also true of the other four classes. One can readily see that if $\{w_1,\ldots,w_n\}$ is the set of variables occurring in any global specification then these may be passed around as parameters to all functions using any of the parameter mechanisms, copy, reference or name. Renaming of local variables may be required in the case where a name is used as a local in one function and specified as global in another. Since each mechanism is capable of returning parametric values to the earlier environment, the result is assured. For P_{Rn} the problem of returning values to an earlier environment may be solved by using the construction of 2.2.1 and the relevant renaming. From these comments we have,

Theorem 2.2.3: Augmenting the classes P_{Rc}, P_{Rr}, P_{Rn}, and P_{Rj} with global variables does not enlarge the class of functionals computed over the corresponding unaugmented class.

Although adding global variables does not enlarge the classes defined by the various parameter mechanisms, globals are extremely useful in reducing the complexity of our constructions. We will use them extensively, but our usage may, at times, become a bit imprecise in that we may use globals without actually mentioning that the class under consideration has been augmented with them. To be completely precise we should also prove

that a class which has been augmented with markers (introduced below) is equivalent to a class augmented with markers and globals. This should be obvious and we use this result without further justification.

Schemata are commonly augmented with a finite set of markers so as to provide testing facilities which are not dependent upon the interpretation. Given the class $M \equiv \{M_1, M_2, \ldots\}$ of distinguishable markers the following two statement forms may be added to the definition of <rec program> given in 1.3. to provide markers for recursive program schemata:

<S> ::= <variable> ← m

| **if** <variable> = m **then** [<label>:]<S> **else** [<label>:]<S>

where m denotes some marker in M. The marker values are independent of interpretation and their semantics should be clear. We adopt the convention that if a marker is passed as an argument to a basic function or predicate or to a main <body> **halt** statement, then Ω will be used. A class P_{Rx} augmented with markers is denoted by P_{RxM}.

In [2] markers were shown to add no power to P_R i.e. $P_{Rv} \equiv P_{RvM}$. This result is shown by using the locator methods introduced in [4]. The same situation arises with P_{RcM} and P_{RrM}, but we need not introduce locators since we can reduce these problems to the result $P_{Rv} \equiv P_{RvM}$.

Theorem 2.2.4: $P_{Rc} \equiv P_{RcM}$, $P_{Rr} \equiv P_{RrM}$.

Proof: It is immediate that the constructions given earlier in this section

apply without modification to P_{RcM} and P_{RrM}. Thus for any schema in either of these two classes we can construct one equivalent to it in P_{RvM}. q.e.d.

The technique to be used when showing that normal evaluation is equivalent to call by reference is to effectively implement the definition of normal evaluation in a call by reference schema. The normal evaluation strategy postpones evaluation of actuals until the formals are required. A call by reference evaluation strategy can be made to correspond if one establishes a communication mechanism between the calling environment and the called function in such a way that the called function can request the evaluation of parameters. In particular, the calling environment passes none of the actual parameters initially, only *indicators* that none have been passed. When the called function requires a parameter it tests to see if the parameter was passed. If it was, the value is used. If not, a *request* indicator is set and the called function halts. This indicates to the caller which parameter is required. It is evaluated and the function called again with this evaluated parameter as an argument. Finally, the calling environment continues calling the function as long as it receives requests for parameters. If a call ever returns with no request for parameters, the result has been computed.

Theorem 2.2.5: $P_{Rn} \equiv P_{RrM}$.

Proof: ($P_{Rn} \leq P_{RrM}$) Let $S_1 \in P_{Rn}$ be a completely labeled schema and let n be the maximum number of parameters to any auxiliary function defined in S_1.

Define for each <body> 4n variables which do not occur elsewhere in the <body>:

$si_1, \ldots, si_n$ called the ***saved indicators***

$sv_1, \ldots, sv_n$ called the ***saved values***

$ci_1, \ldots, ci_n$ called the ***calling indicators***

$cv_1, \ldots, cv_n$ called the ***calling values***

Define S_2 from the following procedure. For the i^{th} auxiliary function call in the <body> of the form:

$$L: t \leftarrow G(\langle term\rangle_1, \ldots, \langle term\rangle_{RG})$$

substitute the statement,

L: **begin**

$si_1 \leftarrow \Omega; \ \ldots; si_{RG} \leftarrow \Omega;$

$sv_1 \leftarrow \Omega; \ \ldots; sv_{RG} \leftarrow \Omega;$

$ci_1 \leftarrow \Omega; \ \ldots; ci_{RG} \leftarrow \Omega;$

L_i: $cv_1 \leftarrow sv_1; \ \ldots; cv_{RG} \leftarrow sv_{RG};$

$t \leftarrow G'(ci_1, cv_1, \ldots, ci_{RG}, cv_{RG});$

if $ci_1 \neq si_1$ **then begin** $si_1 \leftarrow M$; $sv_1 \leftarrow \langle term\rangle_1$; **goto** L_i **end**

else if $ci_2 \neq si_2$ **then begin** $si_2 \leftarrow M$; $sv_2 \leftarrow \langle term\rangle_2$; **goto** L_i **end**

. . .

else if $ci_{RG} \neq si_{RG}$ **then begin** $si_{RG} \leftarrow M$; $sv_{RG} \leftarrow \langle term\rangle_{RG}$; **goto** L_i **end**

else;

end

where: L_i is a label not used elsewhere in the <body>

M is a marker from $\boldsymbol{M}$ and

G′ is a new function corresponding to G defined below.

When every occurrence of an auxiliary function assignment has been replaced with the above statement, eliminate all instances of

if $ci_i \neq si_i$ **then** $\langle S \rangle_1$ **else** $\langle S \rangle_2$

by replacing them with

if ci_i = M **then**

if si_i = M **then** $\langle S \rangle_2$

else $\langle S \rangle_1$

else

if si_i = M **then** $\langle S \rangle_1$

else $\langle S \rangle_2$

which will restore the schema to legal syntactic form for P_{RrM}. Next, for each auxiliary function definition,

$G(u_1,u_2, \ldots ,u_{RG})$:<body>

such that every statement

L: <S> of the <body> where S is

(i) an **if** statement

(ii) an assignment statement such that

(a) there is an occurrence of a formal on the left hand side

(b) there is an occurrence of a formal as a "first level" actual, i.e. u_i is not an actual to an actual.

such that S has as its first occurrence of a formal reference (searching left

to right) the formal u_i, then the <body> of G′ is defined by replacing all instances of such statements by

L: **if** ui_i = M **then** <S′>

else begin $ui_i \leftarrow M$; **halt**(Ω) **end**

where <S′> is <S> with all instances of u_i replaced by uv_i. Clearly, this last construction is applied iteratively to formals of G as long as formal references remain in <S′>. The <body> just defined is prefixed with

$$G(ui_1,uv_1, \ldots ,ui_{RG},uv_{RG}):$$

where the ui_i and uv_i are new variables not occurring elsewhere in the schema S_1. The resulting schema, S_2, is obviously in $P_{R\neg M}$ and as we now show, equivalent to S_1.

First observe that for every statement in S_1 there is a corresponding (but more complicated) statement in S_2. Let $L_1,L_2, \ldots$ be the sequence of labels describing the behavior of S_1, in some interpretation, then S_2 describes the same behavior in that interpretation until a formal u_k of a function G (called at the r^{th} statement) is referenced at the s^{th} statement of S_1. At this point,

$$si_k = sv_k = ci_k = cv_k = ui_k = uv_k = \Omega.$$

Since evaluation proceeds left to right, u_k must be the leftmost formal occurring in the statement of S_1. Thus, the corresponding statement of S_2 tests ui_k, finds it has value Ω, assigns it a marker value and **halts**. In the calling <body>, the calling indicator ci_k = M and the saved indicator $si_k = \Omega$. Thus $si_k \neq ci_k$, which causes the saved indicator si_k to be marked and the saved value sv_k to be assigned the value of $\langle term \rangle_k$. Meanwhile, the

interpreter for S_1 has discovered that u_k has not been referenced before, returns to the calling environment, evaluates the corresponding actual ($\langle term \rangle_k$), assigns it to u_k. At this point, S_1 and S_2 have realized the same semantics. However, S_1 now begins executing G' all over again which implies that the statements with indices r to s will be repeated. When the reference to the formal uv_k is encountered again at statement 2s - r, $ui_k = ci_k = M$ holds, the test will pass and <S'> will be executed. At this point S_1 and S_2 are in register again and execution proceeds in parallel until the next reference. The induction may be carried out in a straight forward manner for which the following facts are useful:

(a) Once a variable is assigned a marker, that name is never changed back to Ω until after a call has been completed and the next one is about to start. Thus, there can not be a repeated evaluation of an actual.

(b) For each repeated call to G', the parameters are always initialized to values of the evaluated actuals and so the behavior of G' must be identical up to the reference of the most recently evaluated formal.

The reader can fill in the details of the induction.

($P_{RrM} \leq P_{Rn}$) The techniques required to show that $P_{RrM} \leq P_{Rn}$ have already been used in earlier proofs, namely, eliminating markers (Theorem 2.2.4) and restoring parameters using repeated calls (Theorem 2.2.1.). These together with coercing all formals at the beginning of the execution of a auxiliary function body, are sufficient to yield the result and the details are left to

the reader. q.e.d.

The above construction separates the use of the markers from the domain objects sufficiently well so that were the data objects to include markers, the construction would still be correct. Thus we have,

Theorem 2.2.6: $P_{RrM} \equiv P_{RnM}$.

Proof: Immediate.

Corollary 2.2.7: $P_{Rn} \equiv P_{RnM}$.

2.3. Summary and Discussion.

The results from the preceding sections may be summarized as follows:

> ***The recursive procedure parameter evaluation mechanisms of call by value, call by copy, call by reference and normal evaluation are functionally equivalent in the sense defined above. Furthermore, these parameter evaluation mechanisms are not functionally enhanced by addition of global variables or by addition of a finite set of markers.***

Parameter evaluation postponement cannot be presented as an explanation of the apparent difference between call by value and call by name. The normal evaluation strategy provides this delaying feature, but no more functionals are computed. In the next chapter call by name will be treated in depth ard its other property, multiple evaluation, will be seen to

be advantageous.

The results of this chapter have all been constructive and so we can ask how practical are the constructions? For example, is it reasonable to consider a direct translation between programs in a language with parameter evaluation of type x into a base language with parameter evaluation of type y? (By direct translation, we mean mapping parameter evaluation into the base language structure without resorting to the use of other data structures to form a new "run time environment.") This would not be prudent since the efficency degrades considerably in our constructions between P_{Rv} and P_{Rc} and between P_{Rr} and P_{Rn} as well as for some cases involving global variables. However, there might be more efficient constructions, though we conjecture that they won't be *much* better. Of course, we are interested in how much one can compute and not in how long it takes. But the proofs do suggest that although these parameter evaluation mechanisms are indistinguishable based on the funcionals they compute, there may be complexity arguments which can provide a distinction.

In the comments preceding definiton 2.1.6, it was observed that call by value had been shown to be "weaker" than the normal evaluation mechanism. This appears to contradict our results showing that these two classes are equivalent. However, there is no contradiction when we understand what the results actually exhibit.

In [3] it is reported that for a continuous functional τ, the computed function F_c using the substitution rule C is no more defined than the least fixed point of τ. The normal rule computes the least fixed point of τ, but call by value (the left-most inner-most rule) computes a function strictly less defined than the least fixed point of τ [15]. Presented according to our development, a sample functional which is less defined than the least fixed point would be given by:

(x,y): z ← F(x,y); **halt**(z)

F(u,v): **if** u = 0 **then** w ← 1 **else** w ← F(u - 1,F(u - v,v));

halt(w)

This instantiated schema, when x = 1 and y = 0, is undefined if interpreted as a call by value schema while the least fixed point function of these two arguments is 1.

The difficulty, of course, is that call by value "gets stuck" evaluating F(u - v,v) when the value is not required for the computation. The normal evaluation mechanism avoids this difficulty. Hence, the sense in which call by value is "weaker" is that when both methods are applied to the same schema, call by value may diverge evaluating unused parameters while the normal rule will not. Clearly, call by copy and call by reference are "weaker" in the same sense, since they would diverge on this same schema. But our results indicate that the classes of functionals defined are the same for both mechanisms. Hence, there exists a functional which, when interpreted as call by value, avoids these unnecessary parameter evaluations.

Furthermore, we can construct such a schema.

An interesting point to note is that our constructions, as presented, have the property that for a schema $S \in P_{Rn}$ the constructed schema $S' \in P_{Rv}$ is such that $Val_v[S',I] \equiv Val_n[S',I]$ for all I where $Val_x[S,I]$ is the value of the schema S in interpretation I with parameter evaluation method x. This suggests that if our results were suitably reformulated, call by value would be a fixed point rule for the translated schemata.

3. Call by Name.

3.1. The Class P_{Rj}

Having learned that the classes of functionals defined by recursion with call by value, call by copy, call by reference and the normal evaluation mechanism are all equivalent, we now consider the class of schemata employing call by name, P_{Rj}. Although one's intuitive feeling is that call by name provides more facility than the mechanisms considered earlier, the proof methods of the last chapter indicate that apparently substantial differences in evaluation strategies can be absorbed into program structure. In this section our intuition will be verified as we show that recursion with call by name defines a larger class of functionals than the other mechanisms do.

To understand the significance of this result, we must realize that in [18] and [4], (call by value) recursion has been shown to be weaker than other programming mechanisms: (call by value) recursion with nondeterministic control and P_{β}, respectively. The conclusion was that recursion is weaker than these other facilities. But one functional which (call by value) recursion cannot compute will be shown (see 3.2.1) to be computable by recursion with call by name. Therefore, the weakness described in [18] and [4] of (call by value) recursive functions is not due to limitations of the basic recursive mechanism, but rather it is due to the way in which parameters are evaluated.

Knowing that P_{Rj} is stronger than the classes considered before, it is natural to ask how much stronger it is. In particular, in [4] it is shown that the above mentioned classes are universal, (see 1.3 for a discussion of the notion of universal.) In the next section we show that P_{Rj} augmented with a single marker, P_{RjM}, is universal. However, unlike the weaker mechanisms where markers can be effectively eliminated, we show that there can be no effective way to eliminate the marker of P_{RjM} schemata to realize schemata in P_{Rj}. This does not imply that P_{Rj} is not universal, since (as Constable and Gries show) nonconstructive means may be used to show equivalence in this circumstance. However, whether P_{Rj} is universal or not is still open and the last section discusses the difficulties of answering this question.

Recall that when the definition of the call by name class was given in 2.1.8, we commented that we could justify our decision to ignore the entire statement when an erroneous left hand side was encountered. If the semantics are prefered of evaluating both sides and then ignoring the results, then replace every assignment to a formal x:

$x \leftarrow G(\ .\ .\ .\ )$

with the statement

begin $w_1 \leftarrow G(\ .\ .\ .\ );\ w_2 \leftarrow x;\ x \leftarrow w_1$ **end**

where w_1 and w_2 are new variables. Clearly, the proper semantics are realized.

On the other hand, if we had chosen the default to be the evaluation

of both sides instead of ignoring the whole statement, then we could realize the same semantics by constructing a new schema as follows. Begin with the main <body> and for every function call

$$G(\langle term\rangle_1, \ldots, \langle term\rangle_{RG})$$

where simple variables are in the parameter positions i,j, . . .,k, substitute a new call

$$G_{i,j,\ldots,k}(\langle term\rangle_1, \ldots, \langle term\rangle_{RG})$$

where $G_{i,j,\ldots,k}$ is a new auxiliary function such that all assignments to formals ***not*** in positions i,j, . . .,k are deleted. Using the fact that formals in positions i,j, . . .,k correspond to identifiers, modify all calls in $G_{i,j,\ldots,k}$ similarly. Using reasoning analogous to that of 2.2.2, the construction must terminate and the resulting schema must have the semantics of ignoring any statement in which an assignment is made to an illegal left hand side. Hence, it makes no difference which interpretation we employ.

3.2. The Strength of P_{Rj}.

To show that call by name is stronger than the mechanisms discussed previously, we will exhibit a functional which can be defined by a P_{Rj} schema but for which there exists no schema in, say, P_{Rv} capable of specifying that functional. The functional was first described by Paterson and Hewitt [18] and has since become known as the "leaftest" functional [2]. It can be stated as follows:

$$\text{leaftest}[p,r,l](x) = \begin{cases} u(x) & \text{if there exists } u \in \{r,l\}^* \\ & \text{such that } p(u(x)) \equiv \text{true.} \\ \text{diverge} & \text{otherwise.} \end{cases}$$

where r and l are basic functions and p is a basic predicate.

In words, leaftest requires a search for a domain element satisfying some predicate where the domain may be thought of as a binary tree. Hence, if x is a domain element (node) then l(x) and r(x) are domain elements (left descendent and right descendent, respectively.)

In [18] it is shown that there is no schema in R (recursive equations, a la LISP) which can implement this functional. The reader should consult the proof in [18] for all of the details, but, intuitively, the "failure" of the recursive equations is that they can scan only a bounded number of nodes on any given ply and an interpretation may be found for which some nodes do not get tested. This result is relevant, since in [4] it was shown that R is equivalent to P_{Rv}.

Theorem 3.2.1: $P_{Rv} < P_{Rj}$.

Proof: That P_{Rj} contains P_{Rv} is immediate since all of the formals can be coerced to locals at the beginning of each function in the schemata of P_{Rv}. These locals may then be used in lieu of the formals and the resulting schemata, interpreted in P_{Rj}, are in keeping with the definition of the call by value evaluation mechanism.

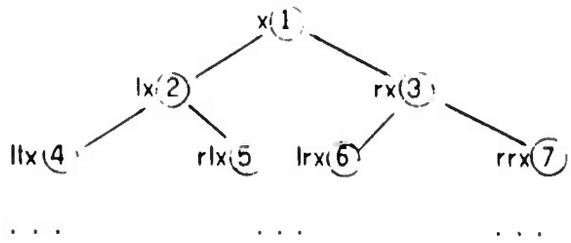
It is now sufficient to give a schema S in P_{Rj} which specifies the leaftest

functional. The following is such a schema; it performs a left to right breadth first search of the binary domain. Note that the function calls never return unless a "true" node is found. The labels L1 through L4 are included only to facilitate subsequent discussion.

```
(x): tail ← Ω; u ← leaftest(x,base); halt(u)

leaftest(node,head): global tail;
                     if p(node) then halt(node) else
                                    begin
                          L1:       tail ← l(node);
                          L2:       nextnode ← head;
                          L3:       tail ← r(node);
                          L4:       loc ← head;
                                    temp ← leaftest(nextnode,queue(loc,head));
                                    halt(temp) end
queue(loc,pred):                    temp ← loc;
                                    loc ← pred;
                                    halt(temp)
base:               global tail;
                    halt(tail)
```

To understand the construction, we number the nodes of the binary domain as follows:



The initial call provides the element x and the function ***base*** as actuals. The reader can follow the execution of the base step. For the induction step, consider the n^{th} call (n - 1^{st} recursive call) of leaftest. The formal ***node*** will correspond to the n^{th} node in the tree and ***head*** will correspond to n-1 nested calls of the function *queue*,

$$queue(loc,queue(loc,queue(\ .\ .\ .(queue(loc,base))\ .\ .\ .)))$$

If the n^{th} node is not the node of interest (i.e. p(node) ≡ false) then the variable ***tail*** is assigned the left descendent of the n^{th} node in statement L1. At L2 the mnemonic significance of the symbols becomes clear. The formal ***head*** corresponding to the nested calls of *queue* is evaluated. In the evaluation of the outer-most call of *queue*, the local is temporarily saved and the next inner-most call of *queue* is evaluated, etc. Finally, ***base*** is evaluated in the most deeply nested environment. ***base*** returns the value of ***tail*** to the most deeply nested instance of *queue*, which stores it and returns its previously stored value, etc. Finally, the outer-most instance of *queue* receives the value saved by the next outer-most instance of *queue*. *queue* returns its stored value as the value of ***head*** which, as we see momentarily, is node n+1 and it is assigned to the ***nextnode*** local of leaftest for testing

on the subsequent call. Thus, our untested values are imbedded in the parameters with a queue-like discipline.

In L3 the right descendent is prepared for substitution into the parameter list and L4 makes the substitution and receives the *head* (node n+2 this time) in the manner just described. This is saved in the local allocated in this environment and the entire operation applied to the $n+1^{st}$ node. Evidently, the descendents of every node tested are generated in the order of our numbering (L1,L3), saved (L2,L4), and each call tests one saved node. q.e.d.

Although the queue type mnemonics have been used only to aid our exposition, the schema S of 3.2.1 does exhibit the way in which queues may be constructively imbedded into P_{Rj} schemata. Brown [1] has studied queue-like data structures and independently observed that a queue could compute the leaftest functional. Of course, we are restricting our study to parameter evaluation, but the following result is worth mentioning so as to correlate our work with [1]. The proof requires the methods just exhibited.

Assertion 3.2.2: The class P_{Rj} computes all of the functionals computed by a class P augmented with an arbitrary number of first-in first-out queues with no "queue empty?" test available.

It is not known whether these two classes are equivalent.

3.3. Universality of P_{RjM}.

The call by name parameter evaluation mechanism is stronger than the other mechanisms considered thus far. However, before considering how much stronger P_{Rj} is, we must analyze this class augmented with markers (P_{RjM}). This class will be shown to be universal by constructing, for every schema in $P_{(n,1)}$, an equivalent schema in P_{RjM}, e.g. $P_{(2,1)}$ is the class of program schemata augmented with two push-down stores and a single marker. There is no "pds empty?" test available for push down stores in the class $P_{(n,1)}$ and pop(PD,z) leaves z unchanged when the pds PD is empty. For $n > 1$, $P_{(n,1)}$ is known [4] to be universal*.

The technique to be used to show that $P_{(n,1)} \leq P_{RjM}$ is analogous to the construction used in 3.2.1 except that we will imbed a pushdown stack in the parameter instead of a queue. Before actually presenting the construction, we show a sample translation of a schema $S_1 \in P_{(1,1)}$ into a schema $S_2 \in P_{RjgM}$.

Example:

```
(x):  L1: y ← f(x);
      L2: push(PD,y);
      L3: pop(PD,z);
      L4: if p(x) then L5: halt(z) else L6: goto L1;
```

---

* $P_{(1,\theta)} \equiv P_{Rv}$ is also shown in [4]. See also [2] for additional results regarding push down stores.

Schema $S_1 \in P_{(1,1)}$. The schema has been completely labeled for comparison to schema $S_2 \in P_{Rjgh}$:

(x): ident ← Ω; y ← Ω; z ← Ω; t ← L1(null); **halt**(t)

L1(pds): **global** x,y; y ← f(x); t ← L2(pds); **halt**(t)

L2(pds): **global** y; save ← y; act ← M; t ← L3(stack(pds,act,save)); **halt**(t)

L3(pds): **global** ident,z; ident ← z; z ← pds; t ← L4(pds); **halt**(t)

L4(pds): **global** x; **if** p(x) **then** t ← L5(pds) **else** t ← L6(pds); **halt**(t)

L5(pds): **global** z; **halt**(z)

L6(pds): t ← L1(pds); **halt**(t)

stack(p,a,s): **if** a = M **then begin** a ← Ω; **halt**(s) **end**

else begin t ← p; **halt**(t) **end**

null: **global** ident; **halt**(ident)

The statements of the original schema have been translated into separate functions with each one responsible for performing the activity of its correspondent as well as calling the next sequential statement function. No function returns until all computation has been completed. (L5 is first to **halt** in the example). In functions corresponding to push statements, a pair of local variables, *save* and *act*, is allocated. *save* retains the value of the pds element while *act* is set to the marker value to indicate that the value has not been used. A value is retrieved (popped) by invoking the pds formal parameter (e.g. L3 above). This forces control to return to the environment which most recently performed a push operation. The function *stack*,

executed in that environment, tests to see if the value in ***save*** has been used. If not, the value is returned and the marker is set. If so, then the formal ***pds*** is again invoked causing a return to the next earlier push environment. If the pds is empty the function ***null*** will eventually be executed. It returns the value which will result in a null operation. Note that the use of the marker is such that no confusion can result if pds elements are also markers.

We now give the construction just illustrated. We require that each syntactic statement in the schema S_1 in $P_{(n,1)}$ be labeled. If L_i is a statement then the ***successor(s)*** is the statement(s) which follows in the sequential control flow. Clearly, **if**s have two, **halt**s have none.

Construction 3.3.1: Let S_1 be a schema in $P_{(n,1)}$ such that each syntactic statement has been labeled. The following steps are required to construct a schema S_2 in P_{RjgM} which is equivalent to S_1.

(i) Let L_i be a labeled statement other than a push or pop:

L_i: <S>

Then an n parameter recursive function is defined as follows:

$L_i(pds_1, \ldots, pds_n)$: **global** $w_1, w_2, \ldots, w_r$;

<S>; $t \leftarrow L_j(pds_1, \ldots, pds_n)$;

halt(t)

where: $w_1, \ldots, w_r$ are the variables used in <S>

L_j is the successor to L_i

t is simple (local) variable not used in <S>.

Actually, we have given only the schematic form for the assignment expression. Since the **begin**s may be ignored (given that the successor has been chosen properly) and we have given examples of the auxiliary functions corresponding to the other statement forms above, we leave it to the reader to construct the functions in the case the statements are other than assignment, push or pop.

(ii) For each push statement of the pds PD_k:

L_i: push(PD_k,v)

define a function

$L_i(pds_1, \ldots, pds_n)$: **global** v; act ← M; save ← v;

t ← $L_j(pds_1,\ldots,stack(pds_k,act,save),\ldots,pds_n)$;

halt(t)

where: t, act and save are new local variables

M is a marker in *M* (see 2.2)

L_j is the successor to L_i.

(iii) For each pop statement of the form:

L_i: pop(PD_k,z);

define a function

$L_i(pds_1, \ldots, pds_n)$: **global** ident,z; ident ← z; z ← pds_k;

t←$L_j(pds_1, \ldots, pds_n)$; **halt**(t)

where:

ident is a new identifier, the same one to be used for all

pop statements. *ident* is the value returned by the function ***null*** to guarentee that pop of an empty stack results in the identity operation.

(iv) Define the two following auxiliary functions:

stack(p,a,s): **if** a = M **then begin** a ← Ω; **halt**(s) **end**

else begin t ← p; **halt**(t) **end**

(v) Construct a schema S_2 in P_{RjgH} using the following main program augmented with the functions defined in (i)-(iv) above.:

$(v_1, \ldots, v_t)$: ident ← Ω; u_1 ← Ω; . . .; u_s ← Ω;

t ← L_1(null, . . .,null); **halt**(t)

where: $v_1, \ldots, v_t$ are the input variables of S_1

$u_1, \ldots, u_s$ are the simple variables used in S_1, (since, precisely speaking, **global**s must be defined in the main program.)

ident is the identifier introduced in (iii) above

L_1 is the first statement of S_1

and t is a new simple variable.

Thus recursive functions augmented with a single marker and employing call by name parameter evaluation can simulate any number of push-down stores. The universal class P_{AL} [4] is capable of simulating recursive functions using the stack model of Dijkstra. This method, used for ALGOL compilers, has been adequately described in the literature [19] and need not concern use here. All that is required in our later proofs is that P_{RjH} is effectively equivalent to P_{AH}. These comments may be summarized as:

Theorem 3.3.2: For $n > 1$, $P_{RjM} \equiv P_{(n,1)}$.

3.4. Necessity of Noneffective Marker Elimination for P_{RjM}.

In this section we establish that if markers cannot be effectively removed from schemata of P_{RjM} to yield schemata in P_{Rj}. The method used is essentially the one used by Constable and Gries[4] to show that P_R is not effectively universal. We require the following definition in order to show that P_{Rj} cannot be constructively equivalent to the universal class P_{RjM}.

Definition 3.4.1 [4]: Let S be a completely labeled schema with n predicates. Then the ***behavior*** of S is the sequence of statement labels of S in the order they begin executing. For any list v of length n chosen from {true,false}, the ***v-autonomous behavior*** of S is the behavior defined by S assuming that for $1 \leq i \leq n$ the truth value of predicate p_i is v_i for all values of its arguments. Clearly, any schema defines at most 2^n different v-autonomous behaviors.

In [4] it was shown that the v-autonomous behavior of P_{AM} is independent of the interpretation of the domain, the input variables and the function symbols. It is also shown that it is undecidable whether the v-autonomous behavior of P_{AM} is finite or not (see 4.2.3 for a similar argument). Clearly, any class effectively equivalent to P_{AM} must also have these two properties. From the last section we know that for $n > 1$

$P_{RjM} \equiv P_{(n,1)}$ is constructive and from [4] we know that for $n > 1$, $P_{(n,1)} \equiv P_{RM}$ is constructive. Thus:

Lemma 3.4.2: It is undecidable whether the v-autonomous behavior of a schema $S \in P_{RjM}$ is finite or not.

However, we shall show that it is possible to determine whether the v-autonomous behavior of a schema $S \in P_{Rj}$ is finite.

Lemma 3.4.3: Let S be a schema in P_{Rj} then it is decidable whether or not the v-autonomous behavior of S is finite.

Note that contrary to the usual case with constant predicates, it *is* possible for schemata in P_{Rj} to execute the same statement twice without entering a loop. However, it is only possible if the statement is a function which is used as an actual parameter and whose evaluation is required for two *different* instances of a formal parameter.

Proof: There are two ways in which the v-autonomous behavior of S can be infinite: a cycle within one <body> or an infinitely regressive sequence of auxiliary function calls. Cycles of the first type can be determined by analyzing the control flow of each <body> using the values v_i whenever the predicate p_i is encountered and ignoring all function calls. If any instruction in a <body> is repeated, a cycle exists and if the main <body> is cyclic then

the v-autonomous behavior is necessarily infinite.

Suppose the main <body> is noncyclic and rewrite the schema with all cyclic bodies deleted and all unreferenced instructions deleted from noncyclic <body>s. From the remaining schema, construct a context free grammar G with sentence symbol S as follows. First assign a unique number to each function *call* in the schema. Define the nonterminals to be new symbols $V_n = \{S, F_1, \ldots, F_r, U_1, \ldots, U_s\}$ where r is the number of function calls and s = kr if k is the maximum number of formals to any auxiliary function. The terminals are defined to be V_t = {all symbols in the schema}. Generate the productions P as follows. If

$$G^i(\langle term\rangle_1, \ldots, \langle term\rangle_{RG})$$

is the i^{th} auxiliary function call of the schema, replace it with the symbol F_i and generate productions:

$$F_i \rightarrow \beta$$

$$U_{i1} \rightarrow \langle term\rangle_1$$

$$\ldots$$

$$U_{iRG} \rightarrow \langle term\rangle_{RG}$$

where β is the <body> of G with all auxiliary function calls replaced with nonterminals and with all occurrences of the j^{th} formal replaced with U_{ij}. (If the <body> is null because it was found to be cyclic, set $\beta = F_i$.) Add the production

$$S \rightarrow \beta$$

where β is the main <body> with all function calls replaced with nonterminals.

L(G) defines a simple context free language such that if L(G) is empty the v-autonomous behavior of S is infinite and if L(G) is finite then the v-autonomous behavior is finite. (The language cannot be infinite since each nonterminal has at most one right hand side in the production set.) Emptiness is decidable by well known methods [10] and is, indeed, a trivial matter for these grammars. q.e.d.

Theorem 3.4.4: There cannot exist an effective procedure which finds for any $S_1 \in P_{RjM}$ a corresponding $S_2 \in P_{Rj}$ such that $S_1 \equiv S_2$.

Proof: Apply lemmas 3.4.2 and 3.4.3. q.e.d.

Our problem of exactly locating the position of the P_{Rj} class has been made a bit more difficult since now we must use nonconstructive methods.

3.5. Discussion.

The question of whether or not P_{Rj} is universal is still open. We know that it is "close" to being universal since the addition of a single marker is sufficient to make it universal. However, there can be no effective procedure for translating all schemata of P_{RjM} into equivalent schemata in P_{Rj}. This phenomenon, of having a class "nearly" universal but not knowing if it is, has been encountered by other investigators. Brown, Gries and Szymanski [2] have the same problem with

P(2b,0) and Brown [1] has a similar difficulty with P_q*. These problems are closely related to the P_{Rj} problem, e.g. if P_q was known to be universal, then methods used earlier in this chapter (3.2) would permit simulation of the queues.

The nonconstructive technique used by Constable and Gries in [4] to show that P_R is universal could also be used for P_{Rj} if it were possible to enumerate the Herbrand Universe. (See [4] for a description of how the Herbrand Universe is used.) In particular, if $u_1, \ldots, u_r$ and $f_1, \ldots, f_s$ are, respectively, the input variables and basic function names of $S \in P_{R,M}$, we wish to construct a $T \in P_{Rj}$ which performs an enumeration of all strings e, where

$$e \in \{u_1, \ldots, u_r\} \text{ or } e = f_i(e_1, \ldots, e_{Rf}) \; 1 \leq i \leq s$$

If this were possible, we could *show the existence* of a schema $S' \in P_{Rj}$ such that $S \equiv S'$. The "difficulty" with finding such a schema T is in organizing the breadth first search so that all terms are treated uniformly and all terms are included. They must be treated uniformly since there are no markers or counters available to impose a substructure on the previously generated values. Clearly, for monadic functions and predicates, the leaftest functional (of 3.2) is an adequate strategy. But it does not seem to generalize to polyadic functions. Several other candidate organizations have been found for T, but the proof that any of these is correct requires number theoretic

---

* P(2b,0) is the class P augmented with 2 push-down stores (with "pds empty?" tests) and no markers. P_q is the class P augmented with queues, for which no "queue empty?" test is available.

results which are themselves open problems. If this is the wrong conjecture and there is actually no way to do this enumeration, then judging from the diversity of the candidate organizations available, proving that none of them works will be difficult indeed.

Finally, note that there is a possibility that P_{Rj} could be shown to be universal by constructive means. Specifically, our results do not deny the possible existence of a translation from each schema $S \in P_R$ to an equivalent schema in P_{Rj}, since the v-autonomous behavior problem is decidable for both classes. The difficulty in actually doing such a translation is in simulating the storing of indices. In the next chapter we investigate P_{Rq}, a generalization of the call by name class. We will present a construction for translating schemata in P_{Rq} into schemata in P_R. The problems of storing indices will be made clear there.

4. Parameter Evaluation Extensions

4.1. The Constituent Parts of Parameter Evaluation.

In this section we abstract the notion of parameter binding so as to understand what operations are required to make the actual - formal correspondence and to understand the consequences of choosing a particular way of implementing these operations. This development will then suggest ways of generalizing and extending the mechanisms considered earlier. In subsequent sections we investigate the classes defined by these generalized mechanisms.

We develop our model from the following point of view. Suppose we are given an interpreter for the basic class B of recursive program schemata (see 1.3). Such an interpreter defines all of the mechanism required to interpret a schema in L(G) except for parameter evaluation. In our development we will establish what operations are required for the interpreter to evaluate parameters using a specific strategy. In essence we will *parameterize* the basic interpreter so that when it is instantiated with the proper functions, it defines an interpreter for a certain parameter evaluation mechanism. Our task is to abstract the constituent parts of parameter evaluation (i.e. to define the formals to our interpreter). We also present the functions for these parts (actuals) which realize the various evaluation mechansims discussed thus far.

At this point we could introduce an abstract interpreter for our basic class of schemata, defined in some language (e.g. Vienna Definition Language). However, such mechanism is not required since the greater part of such an interpreter doesn't concern us and the points at which our constructions must interface with the rest of the interpreter are few and simply described.

The interpreter manipulates several *types* of objects, e.g. domain values, identifiers, etc. We will use a simple list-like notation to denote the types of objects of interest to us. Let a pair <A,B> denote any object whose left hand side value is an object of type A and whose right hand side value is an object of type B. The particular objects of the basic interpreter in which we are interested are

$$\langle L,D\rangle, \langle\ ,D^n \rightarrow D\rangle, \langle\ ,E\rangle.$$

The first pair denotes an object corresponding to a variable, i.e. its left hand value is of type location (L) and its right hand value is of type D where D is the domain of interpretation. Basic functions with rank n are denoted by the second term, (there is no left hand value since they cannot be assigned to). The third term represents unevaluated expressions, i.e. auxiliary function calls. There are other kinds of objects used by the interpreter, (e.g. predicates) but they cannot be parameters to functions. The objects denoted above are the only objects which may be used as actual parameters and thus they are the only ones of interest to us.

In parameter binding we are interested in environment formation. Environments provide the correspondence between the syntactic objects and the computational objects which they denote. Thus, an environment, ρ, is a mapping

$$\rho: Id \rightarrow Dn$$

where Dn is the class of denoted objects. We have no interest in the particular deroted objects, only the *types* of the denoted objects, so we write

$$Dn = \{<L,D>, <,D^n \rightarrow D>, <,E>\}$$

to indicate the types of denoted objects of an environment.

Environments change when a function is called and so we suppose that there is a binding functional β in the interpreter which defines a new environment from the current one. β defines environments inductively and to keep these separate, we index them. Thus, the denoted types in the environment of the main <body> are

$$Dn_0 = \{<L,D>, <,D^n \rightarrow D>, <,E>\}.$$

When a function is called from the zero environment, the binding functional associates the actual parameters with the formal names. Thus, the objects of the first environment are,

$$Dn_1 = \{<L,D>, <,D^n \rightarrow D>, <,E>, <L,<L,D>>, <L,<,D^n \rightarrow D>, <L,<,E>>\}$$

where the first three terms are the types of objects specifyable in any environment. The formals are given in the last three terms indicating that formals have left hand values of type location and right hand values which

are of actual object type of the zero environment. Recursive application will, clearly, define denoted objects of larger and larger type.

The binding function only assigns a name to the object which was the actual parameter. We would not have been calling this "parameter evaluation" all this time if parameters weren't being *evaluated* as well as being assigned a name. But evaluation is a poor word since we wish to treat cases where no evaluation takes place. Hence, we hypothesize a translation function, τ, as one of the parametric inputs to our interpreter.

The translation function is used by the binding functional to translate actual objects into values of formal type. The translation function τ has as arguments the actual and the name of the calling envrionment. Thus, the denoted objects in environment i are defined, recursively,

$$Dn_0 = \{<L,D>, <,D^n \rightarrow D>, <,E>\}$$

$$Dn_i = Dn_0 \cup \{<L, \tau(x,i-1) \mid \text{for all } x \in Dn_{i-1}\}$$

which states that the types of the objects in environment i are those objects specifyable in any environment (Dn_0) together with the newly introduced formal objects and these correspond to the translated actual objects of the earlier environment bound to a object of type location.

To see how τ works, consider the translation function for call by value or call by copy. For the call by value mechanism,

$$\tau_v(x,i) \equiv eval(x,\rho_i)$$

where *eval* is the expression evaluation mechanism of the interpreter and ρ_i

is the i^{th} environment. This simply requires that the binding functional evaluate the actual in the calling environment, and thus τ_v maps the actuals into domain elements D. Any formal defined by τ_v is of type <L,D> and thus we have

$$Dn_\beta = Dn_i \text{ for all } i$$

in the case of call by value. Clearly, call by copy has the same translation function as call by value; hence $\tau_c \equiv \tau_v$.

For the other mechansims, the translation function is a bit more complex. For example, one way to describe the translation function for call by reference is,

$$\tau_r(x,i) \equiv \textbf{if } x \neq \text{variable } \textbf{then } \langle L, eval(x,\rho_i)\rangle \textbf{ else } x.$$

The surrogate variable of our definition (2.1.4) is represented by an object of type L. If one used this translation function for call by reference, the denoted objects would be of types

$$Dn_\beta \cup \{\langle L,\langle L,D\rangle\rangle, \langle L,\langle L,\langle L,D\rangle\rangle\rangle, \langle L,\langle L,\langle L,\langle L,D\rangle\rangle\rangle\rangle, \ldots\}$$

and any formal of environment n would have up to n levels of reference values. If this were the definition, the obvious implementation would be quite convenient on a machine with repeated indirect addressing, since such operands correspond exactly to the objects in the above set (on a machine with an infinite memory). But our definition in 2.1.4(i)c required that if the actual is itself a formal, the the reference value is to be passed. We can incorporate this requirement with the following translation function:

$$\tau_r'(x,i) \equiv \textbf{if } x \neq \text{formal } \textbf{then } \tau_r(x,i) \textbf{ else } cdr(x).$$

Clearly, the denoted object types for call by reference become:

$$Dn_0 = \{<L,D>, <\ ,D^n \rightarrow D>, <\ ,E>\}$$

$$Dn_i = Dn_0 \cup \{<L,<L,D>>\} \text{ for } i > 1.$$

The translation function for normal evaluation and call by name evaluation are given by:

$$\tau_n(x,i) \equiv <x,\rho_i>$$

$$\tau_j(x,i) \equiv <x,i>$$

which indicates that the translation function need not perform any modification, but needs only to bind the actual with an environment (for normal evaluation) and to bind the actual with an environment name (for call by name evaluation.) These definitions for τ_n and τ_j correspond to those of 2.1.6 and 2.1.8, respectively. Since our notation provides a method for referring to specific environments, we are not required to specify the arbitrarily deeply nested types. This observation simplifies our subsequent description, and thus we define the equivalent τ_n' and τ_j' as follows:

$$\tau_n'(x,i) \equiv \textbf{if } x \neq \text{formal } \textbf{then } <x,\rho_i> \textbf{ else } cdr(x)$$

$$\tau_j'(x,i) \equiv \textbf{if } x \neq \text{formal } \textbf{then } <x,i> \textbf{ else } cdr(x).$$

The point to note about the translation functions τ_r, τ_n and τ_j, is that they did an incomplete job of translation in the sense that when the translation was made the resulting denoted objects were of different type than the usual local objects. That is,

$$Dn_0 \neq Dn_i \text{ for } i > 0.$$

when β depends on any of the translation functions presented for call by reference, call by name or normal evaluation. Since formals may appear wherever locals can, we must further state, when specifying a parameter evaluation mechanism, how values may be found which are acceptable as local objects.

We find, therefore, that we must further parameterize our basic interpreter with definitions for two other operations, L to find left hand values for formals and R to find right hand values for formals. The simple case is when the mechansim of evaluation is call by value or call be copy, since the translation functions τ_v and τ_c mapped actuals into objects the interpreter could already handle, i.e. $Dn_\theta = Dn_i$. The types of the formal variables are the same as the local variable type, <L,D>. Thus, the same reference mechanism may be used for formals which is used for locals. We then have, for a formal symbol x interpreted in environment ρ,

$$L_v(x,\rho) \equiv L_c(x,\rho) \equiv L(x,\rho) \equiv car(\rho(x))$$

$$R_v(x,\rho) \equiv R_c(x,\rho) \equiv R(x,\rho) \equiv cdr(\rho(x)).$$

where L and R are the reference mechanism for locals which are already specified for the basic interpreter.

The interesting cases are when $Dn_\theta \neq Dn_i$ for $i > 0$. For call by reference, the translation function τ_r assigns formals values of type <L,<L,D>> and so for a formal x in environment ρ, the reference functions become:

$$L_r(x,\rho) \equiv car(cdr(\rho(x)))$$

$$R_r(x,\rho) \equiv cdr(cdr(\rho(x)))$$

which reference the reference value and coerced value, respectively.

For normal evaluation formals defined by τ_n' are of type <L,<a,ρ_i>> where a is the unevaluated actual and ρ_i is the environment of the call. For normal evaluation of a formal x defined by τ_n' in environment ρ the reference functions are given by:

$$I_n(x,\rho) \equiv \textbf{if } cdr(\rho(x)) < D \textbf{ then } car(\rho(x))$$

$$\textbf{else begin}$$

$$eval(car(cdr(\rho(x))),cdr(cdr(\rho(x))));$$

$$\textbf{halt}(car(\rho(x))) \textbf{ end}$$

$$R_n(x,\rho) \equiv \textbf{if } cdr(\rho(x)) < D \textbf{ then } cdr(\rho(x))$$

$$\textbf{else } eval(car(cdr(\rho(x))),cdr(cdr(\rho(x))))$$

To see that these are consistent with 2.1.6, observe that $cdr(\rho(x)) < D$ is true if and only if the formal has been assigned to in the <body> of the call. If it has not been, then it must be evaluated before it is assigned to (according to 2.1.6), and thus, the **else** consequent of I_n. If the right hand side is required, then if the formal has been assigned to then we produce that value. If it has not been assigned to we evaluate it each time it is referenced. This is not a contradiction of our definition of 2.1.6 since in our formalism, the environment ρ_i is passed along and there can be no side effects. Thus, the first evaluation is indistinguishable from the n^{th} and we

need not concern ourselves with which it is except for efficiency reasons*.

Finally, the call by name formals defined by τ_j' are of type <L,<a,i>> where a is the unevaluated actual and i is the calling environment name. The reference function for a formal x in environment ρ is given by,

$$L_j(x,\rho) \equiv car(car(cdr(\rho(x))))$$

$$R_j(x,\rho) \equiv eval(car(cdr(\rho(x))),\rho_{cdr(cdr(\rho(x)))})$$

Note that if the actual does not have a left hand side, L will return nothing and we suppose the interpreter can handle such cases.

The only remaining detail is that of copying back of formal values into actuals for call by copy. This requires the use of the assignment functional of the interpreter and since there is no particular subtlety, we shall ignore it for now.

Note that when the left hand side reference function returns a location other than $car(\rho(x))$ (i.e. other than the element provided by the binding functional) there will be side effects in some environment. Clearly, if we redefine the reference functions for call by reference to be

$$L_r(x,\rho) \equiv car(\rho(x))$$

$$R_r(x,\rho) \equiv \textbf{if } cdr(\rho(x)) \in D \textbf{ then } cdr(\rho(x))$$

$$\textbf{else } cdr(cdr(\rho(x)))$$

---

* Actually, efficiency is the whole reason normal evaluation was introduced in [15]. We could easily specify a "once if ever" evaluation strategy by using the assignment functional of the interpreter, but we choose to avoid the introduction of this extra mechanism here, since the meaning is the same.

we produce an evaluation mechanism which is indistinguishable from call by value.

Excepting the detail concerning call by copy then, we can parameterize the basic interpreter for a particular evaluation mechanism x by specifying the three components:

τ_x the translation function,

L_x the left hand side conversion procedure

R_x the right hand side conversion procedure

The fact that parameter evaluation introduced a new type of value implies that type conversion procedures L and R have to be introduced to translate between formals and other types. These new procedures provide *inter*object translation. Interobject translation is a consequence of choosing τ_x such that the resulting denoted objects are not all in Dn_θ. But not all operations take place between formals and other objects. Some operations take place between formals and other formals. For example, formals can be assigned to formals. Thus, there must also be *intra*object translation. If one can pass an unevaluated function call or formal, why can't such unevaluated objects be assigned to formals within a body. Of course, that can be done and many authors have considered this problem, e.g. Landin, Reynolds, Scott and Stratchey.

We will not, however, address the issue of adding functions as a value type. In the first place, our little model of environment defintion would fall

apart. Secondly, the problem has been studied before, (see [20] for an overview.) Finally, our interest is in classes of functionals and languages as introduced by the above authors are so powerful as to surely be universal (see next section).

However, our inability to show that call by name is universal motivates us to find a generalization which would suggest what the shortcomings of its earlier definition are. The notion of *intra*type conversion is just the mechanism. Towards this end, if we observe that parameter passing for call by name is the assignment of unevaluated functions to formal parameter identifiers, then we could allow all assignments to formal identifiers to be unevaluated. This is analogous to the *quote* operation in LISP, only we will apply it only to formals. The restriction to formals is appealing since it doesn't require any substantial modification of our definitions, as a general *quote* operation would. As we will see in the next section, this is enough.

4.2 The Class P_{Rq}.

In this section we investigate the call by name extension suggested in the last section. The mechanism, which will be entitled call by quote, warrants a few preliminary remarks. It is admittedly pathological in the sense that although it is a extension of call by name, it probably not a viable parameter mechanism for any real language. This is because, when call by quote facilities are provided, one might as well allow functions as

values throughout the language, i.e. as local identifier and function values. Such a language is too large a step for our interests here. Finally, note that the "stack" model which is usually used with call by name no longer applies.*

Definition 4.2.1: The class P_{Rq} of *recursive program schemata with call by quote* is a call by name class of schemata with the additional semantics that *any* assignment to a formal is unevaluated. If the formal does not have a left hand value, the entire statement is ignored. If a formal must be evaluated and its value is a circular sequence of references, the value is Ω.

Example:

(x): y ← G(x); **halt**(y);

G(z): w ← f_1(z); **if** p_1(w) **then** z ← G(w) **else**; **halt**(w)

A sample schema which if interpreted in the class P_{Rq} calls G exactly once from the main <body>. Compare this to its interpretation in P_{Rj}.

Although we do not know if P_{Rj} is universal, if it is not then the following theorem tells us that the generalization in P_{Rq} is sufficient for universality.

Theorem 4.2.2: $P_{Rq} \equiv P_R$.

Proof: Appendix I contains the details of a construction whereby any schema

---

* This may not be entirely correct if one can appropriately modify Fischer's proof [7], which uses call by value, to handle this case.

$T \in P_A$ can be effectively translated into an evidently equivalent schema $T' \in P_{Rq}$. Informally, the translation is straight forward once there is a way to allocate an infinite linear vector and a way to manipulate indices (specifically, to store an index.) We allocate the storage essentially as was done in sections 3.2 and 3.3 as formals to the recursively called instructions of T. To realize subscripts we use the fact that unevaluated expressions can be assigned to formals. Thus we have a *zero* function and an *index1*(x) function. The subscript two would be *index1*(*index1*(*zero*)). To get an array value, control must first return to the earliest environment (i.e. 0 cell of the array). Then, the index is coerced, removing one level of nesting per environment until the zero function is called. This function transfers the value of the local (corresponding to the array cell) to a global temporary formal and destroys itself. Thus there is a fetch from the indexed cell. Assignment works in the analogous way.

Of course, schemata in P_{Rq} can be simulated by a universal class, say P_{AL}. Weisenbaum's paper [23] provides a good presentation of the problems involved. q.e.d.

With P_{Rq} known to be universal, it is reasonable to consider trying to find a translation from schemata in P_{Rq} into schemata in P_{RJ} and thereby showing its universality. This is not, however, an advisable strategy as the following theorem indicates.

Theorem 4.2.3: It is not decidable whether the v-autonomous behavior of P_{Rq} is finite or not.

Proof: Given a Turing machine Tm, we can construct a schema $T \in P_{Rq}$ which simulates the behavior of Tm. The schema in P_{Rq} employs no predicates and thus operates under constant truth assignments. If T halts then Tm halts on blank tapes. Since this property is not decidable, v-autonomous behavior cannot be. The construction is quite envolved and is presented in Appendix II. q.e.d.

Corollary 4.2.4: There cannot exist an effective procedure which finds for any schema $S_1 \in P_{Rq}$ a schema $S_2 \in P_{Rj}$ such that $S_1 \equiv S_2$.

Proof: Apply theorem 4.2.3 and lemma 3.4.2. q.e.d.

Our generalization was too generous. It may be that adding some extra strength to the P_{Rj} class will enable one to build a bridge between P_{Rj} and P_R, but at this point its not clear what such a facility would be.

5. Decisions about Parameter Evaluation.

5.1. Motivation.

The various parameter evaluation mechanisms which we have studied represent different solutions to the same semantic problem: binding values of one environment to names in another environment. As different solutions, they provide differing degrees of convenience at differing costs. For example, normal evaluation might require a minimum of execution time for a small number of references but the test for "already evaluated?" might incur too much overhead for large numbers of parameter references. On the other hand the advantage of being able to return parameter values with call by reference might make it more convenient than normal evaluation. In short, convenience of use and cost of implementation and execution are among the axes along which parameter evaluation mechanisms vary.

Convenience is a personal issue and implementation cost is a highly machine dependent issue. Therefore, we will not treat either specifically. Rather, we will provide some facts for use by the language designers and implementors which may help them to optimize these (and other) conflicting features when choosing a parameter evaluation mechanism.

In particular, we observe that if one could determine "how a parameter is to be used" when compiling a program, one might be able to decide which is the "cheapest" mechanism available that provides the required semantics.

For example, in ALGOL the mechanisms of call by value and call by name must both be available. Since the compiler writer must provide both mechanisms, he might prefer to implement call by name evaluations as call by value in cases where it wouldn't make any difference and would be cheaper. We propose to investigate to what extent questions of this type can be decided and, when they can be, to give procedures for answering them. Armed with this information, we could expect the language designer to specify the parameter evaluation mechanism he deems most convenient while the implementor provides that facility only when absolutely necessary and whenever he can, uses any other mechanism that is cheaper.

5.2. Definitions.

If the reader is familiar with the unsolvability results for in program schemata, especially Luckham, Park and Paterson [14] then he will be skeptical of our chances of establishing any but negative results about optimizations. In this section we present the necessary definitions and then justify the reader's feelings. We will not, however, spend a lot of time providing negative results, but rather in subsequent sections we try to establish decidable, though less complete, optimizations.

Recall that the syntax of our various classes is given by the grammar G (in 1.3.) Let L(G) denote the class of terminal strings of this grammar which we called basic recursive programming schemata. In addition, P_{Rx} is

the class of recursive programming schemata with x parameter evaluation and $Val_x(S,I)$ is the "interpreter" function for the class P_{Rx} evaluating schema S in interpretation I.

Definition 5.2.1: Let P_{Rx} nd P_{Ry} be classes of recursive program schemata defined on strings of L(G) with parameter evaluation mechanisms x and y, respectively. Then the *coincidence class for* x *and* y, C_{xy} contained in L(G) is the class of terminal strings such that

$$C_{xy} = \{S \in L(G) \mid Val_x(S,I) \equiv Val_y(S,I) \text{ for all } I\}$$

Thus the coincidence class for x and y contains exactly those schemata whose interpretation is insensitive to which method of parameter evaluation, x or y, is applied. Since L(G) contains some strings with no parameterized auxiliary functions, C_{xy} is trivially nonempty. On the otherhand, the coincidence class can contain many schemata as is seen in the example.

Example 5.2.2: In section 2.2 we observed that for any schema S, the call by copy interpretation of S is indistinguishable from the call by reference interpretation of S if all simple variables in each auxiliary function call are distinct. All such schemata are elements of the coincidence class C_{cr} for copy and reference.

Of course, the class C_{cr} also contains schemata with duplicated actuals in those cases where the duplication makes no difference.

Let x and y be parameter evaluation mechanisms, then a ***parametrically induced name-value disparity*** describes the situation when some schema is interpreted using the two different mechanisms and the values associated with the names become different in the two interpreters as a result of the parameter evaluation activity. Specifically, suppose that all the names have the same values in the two interpretations at the commencement of one of the parameter binding activities (i) auxiliary function initiation, (ii) format parameter evaluation or (iii) auxiliary function termination. If, at the completion of this activity, there exists a name whose value differs between the two interpretations, then we say the two evaluation methods cause a parametrically induced name-value disparity.

Definition 5.2.3: Two parameter evaluation mechanisms x and y are said to be ***semantically separable*** if:

(i) there exists a schema S and an interpretation I such that $Val_x[S,I] \neq Val_y[S,I]$,

(ii) $Val_x[S,I] \neq Val_y[S,I]$ implies that there exists a parametrically induced name-value disparity.

The definition requires that for two evaluation methods to be separable they must be different and the difference must be manifested at the very least by difference in the state vectors of the interpreters caused by parameter evaluation. (Of course, we consider the case of a value verses no value (as one might get with evaluation postponement) as being a value

disparity.) Each pair of parameter evaluation methods which we have discussed are semantically separable (see Corollary 5.2.5 below for justification.) As an example of a semantically inseparable pair, consider call-by-value (lr) and call-by-value (rl). These are both call by value but differing only in that the former evaluates actuals left to right and the latter evaluates them right to left.

Theorem 5.2.4: Let x and y be parameter evaluation mechanisms which are semantically separable. Then the membership problem for the coincidence class C_{xy} of x and y is not partially decidable.

Proof: By virtue of the fact that the parameter evaluation mechanisms are semantically separable, a schema S exists for which there is a parametrically induced name-value disparity for all interpretations. We may construct for any schema $R \in L(G)$ a new schema S_R whose value is provided by the name whose value diverges according to which method of evaluation is provided. Furthermore, the value assigned to the name will be in the one case the value of R and in the other case some value different from, but dependent upon, R, e.g. h(R) for some basic function h. Such a construction implies

$$S_R \in C_{xy} \iff R \text{ diverges for all interpretations.}$$

The theorem then follows from the fact that the class P of flowchart schemata is properly contained in L(G) and that the problem of deciding if a schema $R \in P$ diverges under all interpretations is not partially decidable (see 1.3). q.e.d.

Corollary 5.2.5: For any distinct pair of parameter evaluation methods, call by value, copy, reference, name and normal evaluation, the memebership problem for the coincidence class for the pair is not partially decidable.

Proof: If two of the methods differ because only one postpones evaluation or assigns back into the actual, semantic separability is immediate. This leaves only the pair <call by copy, call by reference> and the example in 2.1.10 provides the required schema for separability. q.e.d.

Thus, we cannot do perfectly well at identifying when the facilities of a given parameter evaluation mechanism are not fully utilized. The reader might feel as though our use of strong equivalence was unfair and that our result depends upon a peculiarity of strong equivalence. First of all strong equivalence seems quite appropriate when considering techniques to be used for compilers, etc. Secondly, we could modify our definitions to a "weaker" relation and then employ the very strong results of Luckham, Park and Paterson [14] on "reasonable relations". The even stronger results of Itkin and Zwienogrodsky [12] could also be used but both methods would be more involved and the result would most certainly be the same. The task of doing "perfectly well" at optimization is so hopeless as not to be worth such effort. We compromise and try to do "reasonably well" in the next sections.

5.3. Decidability Lemmas.

In this section we will make a rather strong assumption about our schemata and prove several lemmas to be used in the next section. The lemmas will enable us to recognize when the full semantics are used of a particular parameter evaluation mechanism. Such results will enable parameter bindings to be compiled so that the cheapest mechanism is used providing the required semantics.

In [14] Luckham, Park and Paterson introduced the notion of a free program schema. We will use the same notion for our recursive schemata.

Definition 5.3.1: A schema $S \in P_{Rx}$ is said to be *free* if every behavior of S is an execution behavior.

Recall (from 3.4.1) that the behavior of a schema is merely the sequence of statement labels as they begin execution and a behavior is an execution behavior if there is an interpretation which will realize that behavior. This is a very strong requirement. Indeed, in [14] it is proved that freedom is not a decidable property for program schemata and so it cannot be for recursive schemata.

We begin by proving a simple lemma about free schemata. A property is *sensitive* to function calls if the decision whether or not a particular function environment has the property is affected by the functions it calls. Clearly, the property of *halting* is sensitive (in the general case) to function

calls since to determine if a function halts one must know if the functions it calls halt.

Lemma: 5.3.2: For any free schema S in P_{Rx} the following properties are decidable:

whether or not for a formal y of an auxiliary function G

(1) y appears on the lefthand side of an assignment,

(2) y is ever referenced, and

(3) y is not referenced on at least one path.

Proof: It is immediate when the property is insensitive to the evaluation methods. Suppose the property is sensitive to function calls, (in particular, sensitive to the evaluation strategy.) Then, the following outline of a procedure should suffice to recognize the presence or absence of the properties:

(a) For an auxiliary function, G, mark the formals with an X, O or ? depending on whether, from local information, the formal satisfies the property, doesn't satisfy it or its "not known" whether it is satisfied, respectively. The "not known" case will occur when the formal is also an actual to an auxiliary function.

(b) Choose an auxiliary function at least one of whose formals is question marked. Choose a formal, z, and find which auxiliary functions it appears in as an actual and in which positions. Decide whether the properties hold in this function. If *any* of these are

marked X, z is marked X. If *all* are marked 0, z is marked 0. Otherwise, z remains question marked.

(c) Apply (b) until no question marks can be removed, which must occur since new ones are not introduced.

Parameters still marked with a question mark evidently do not satisfy the property and should be marked 0, since they form only closed cycles. q.e.d.

Whether a formal is eventually assigned to or referenced is not enough information. We must know whether that assignment or reference has any effect.

Lemma 5.3.3: It is decidable for a free schema $S \in P_{Rx}$ whether postponing actual parameter evaluation avoids a nonterminating computation.

Proof: A syntactic check can be made to determine which parameter positions are ever filled by auxiliary functions. By lemma 5.3.2 it can be determined which of these parameters is ever referenced. For those that are referenced, the auxiliary functions filling their actual parameter postions can be examined to determine which diverge, since a free auxiliary function, G, containing I statements will diverge in some interpretation if there exists a behavior of G containing more than I statements. For the formals referencing potentially divergent parameters, divergence of the entire schema can be avoided if not all paths through the function reference the formal. Mark all paths on which the formal is referenced (assuming use as an actual to an auxiliary function is not a reference.) From lemma 5.3.2 the remaining

unmarked paths can be marked depending on whether evaluation is avoided in any of the called functions. If there exists an unmarked path to a **halt** then postponement will avoid a nonterminating computation. q.e.d.

In the following lemma we prove that it is decidable whether or not an assignment to a formal changes its value. There is a bit more complexity to it than the earlier proofs, and the following definitions will be useful.

Definition 5.3.4: A ***k - repeated behavior*** of a schema S is a behavior $L_1, \ldots, L_n$ such that there are no indices i0,i1,i2, . . .,ik+1 such that,

$$L_{i0}, \ldots, L_{i1-1}, L_{i1}, \ldots, L_{i2-1}, \ldots, L_{ik}, \ldots, L_{ik+1}$$

is a substring of $L_1, \ldots, L_n$ and

$$L_{i0}, \ldots, L_{i1-1} = L_{i1}, \ldots, L_{i2-1} = \ldots = L_{ik}, \ldots, L_{ik+1}.$$

Thus, a k-repeated behavior is one in which no iteration or recursion cycles on the same path more than k times. Clearly, for fixed k, any k - repeated path through a schema S is finite.

Definition 5.3.5: Let e be any expression in the Herbrand Universe (see 3.5 for def.) such that $e = f(e_1, \ldots, e_{Rf})$. Then the ***generation tree*** for e is the oriented tree formed by placing f at the root and the Rf edges from it (from left to right) leading to the generation trees for $e_1, \ldots, e_{Rf}$, respectively.

Thus, a generation tree for any computed value of a schema has basic function symbols for nodes and leaves of either input variables or Ωs. The

term cycle in the following theorem means either an iteration or a recursion which returns control to the same label.

The crucial fact to note regarding recognizing when an assignment is always an identity assignment is that if it is then the subcomputations must either be constant or the computation of the left and right hand sides must take place "in unison." By "in unison" we mean that a subcomputation for one cannot take place in one cycle and the corresponding subcomputation in a different cycle, for if they did, we could find (in a free schema) different numbers of repetitions for the two cycles to contradict the assumption of equality. Thus, cycling through a loop (iteratively or recursively) must always perform subcomputations for both left and right hand sides.

Lemma 5.3.6: For a free schema S in the class P_{Rx}, it is decidable whether or not an assignment to a formal y of an auxiliary function G changes the value of y.

Proof: We first prove a major subcase: the problem is decidable where S has no identity assignments. Enumerate all 2 - repeated paths through S and decide whether the assignment in question ever changes the value of y. We now show that if for none of the 2-repeated behaviors the value of y is changed, then there can be no behavior $L_1, \ldots, L_n$ in which the value of y is changed. Observe the following facts:

(1) Since there are no identity assignments, if both sides of the assignment in question have a bounded length, then all possible values

of the assignment have already been investigated.

(2) From (1) it follows that if a behavior $L_1, \ldots, L_n$ exists then it must have a subcomputation in a cycle the value of which depends on a value changed in a cycle.

(3) Since there are no identity assignments a value changed in one repetition of a cycle must either be used by the end of the next repetition or never be used.

Suppose a behavior $L_1, \ldots, L_n$ exists and that it is a shortest such behavior. Then there exists a 2-repeated behavior $M_1, \ldots, M_p$ of S and an index i such that $L_n = M_p$ is the assignment in question and $L_{n-i}, \ldots, L_n = M_{p-i}, \ldots, M_p$. Thus, there is a suffix of the 2-repeated behavior which corresponds to the suffix of our supposed counter example. We choose $M_1, \ldots, M_p$ so that its suffix is the largest possible. Let E_1 and E_2 be the generation trees of the left and right hand sides of the assignment in question. Then since the suffixes match, then the top k ply of E_1 and E_2 match for some k. Choose two values e_1 and e_2 in the generation trees of E_1 and E_2, respectively, such that $e_1 \neq e_2$, they fill corresponding parameter positions and all nodes on a path to their respective roots match. These must exists or else E_1 and E_2 are not different. We require that the root nodes of e_1 and e_2 differ. By working backwards from L_n through the behavior $L_1, \ldots, L_n$, we can find where these two values were computed.

From (1) we know that these values had to be computed in one or more cycles. In addition, we can show that they were computed in the same

cycle. Suppose they were computed in separate cycles, then either,

(a) one or the other cycle has been repeated and if repeating the cycle doesn't change the value we can eliminate a repetition, realize the same value and contradict the assumption that $L_1, \ldots, L_n$ is shortest.

(b) one cycle has been repeated and repetition yields a different value for each iteration. If the repetition is greater than two then we can eliminate one or more, realize a different value and contradict the assumption that $L_1, \ldots, L_n$ is the shortest sequence.

(c) the cycle has been repeated 0,1, or 2 times with a different value each time and thus has been discovered in our 2-repeated behavior analysis, contrary to the assumption that none existed.

Those are the only choices with no identity assignments, so we conclude that the two values e_1 and e_2 were computed in the same cycle. But this is impossible since if it were true then we have already discovered the difference. Since the suffixes match, e_1 and e_2 are operands to the same function name f (though maybe not the same instance of the name) then that name must be outside the loop or on the next repetition of the same cycle. If it is outside the cycle then any exit from the cycle is erroneous, including the first. If the instance of f is in the cycle, then the error must be discovered in two repetitions through the cycle since with no identity assignments values computed on one cycle must be used on the next, or never. Thus, two repetitions of the cycle are sufficient to recognize the difference regardless of where the occurrences of f are. Thus the supposed

difference cannot exist.

The effect of identity assignments is to act possibly as a delay line and to postpone the decisions required for the above argument. In particular, for any operand name a change in value can be postponed for a maximum of k iterations of a cycle where k is the number of identity assignments in a loop. Thus, we merely enumerate all k-repeated behaviors where k is the number of assignments in the repeated subbehavior. q.e.d.

Lemma 5.3.7: It is decidable for a free schema S whether or not a change in the value of a formal y of any auxiliary function F affects the value of the schema.

Proof: There is a bound on the amount of simulation required to determine if any output of an auxiliary function G (or main program) is affected by a change of a formal y in F. If there are n variable names (for formals and locals) in G then there are 2^n possible assignments of values which either depend or do not depend upon an actual whose value could be changed by the formal y of F. The *dependency state* is an n element binary vector, one bit corresponding to each variable, describing whether the value of the variable could change as a result of the function call (1) or would be unaffected (0). Initially, the vector is all zeros. Begin simulating G and set the bit corresponding to x if

(1) x is the actual corresponding to y in a call of F,

(2) x is assigned a value which is a function of variables which have

their bits set,

(3) x is assigned two different values in each consequent of a predicate whose parameters include variables which are set.

The vector element corresponding to x is to be set to zero if x is assigned a value which is a function of unmarked variables. All computations are to be simulated with a record kept of the current value of the dependency state for each labeled statement. When a **goto** L is encountered, the current dependency state is compared with those previously in effect when L was executed on this path. If the current dependency state is in the list, then no further execution of the path is required, since nothing new can happen. This fact limits the length of any path which must be considered. **goto**s must be executed if the computation is not finite, and so this process must terminate. If all simulations lead to **halt** statements whose argument x does not have its corresponding bit set, then a change in the formal y does not affect the result of G. In all other cases it does. q.e.d.

5.4. Coincidence Class Decidability.

The lemmas from the last section will enable us to decide the coincidence class problems for free schemata.

Theorem 5.4.1: For free schemata the membership problem for coincidence classes C_{vc}, C_{vr}, C_{vn} and C_{vj} is decidable.

Proof: Apply lemmas 5.3.6 and 5.3.7 for C_{vc} and C_{vr}. Apply lemma 5.3.3 for C_{vn}. For C_{vj} all three lemmas solve half of the problem. The other half is recognizing whether there are side effects when the actual is coerced. If there are, multiple evaluations may be required and values of simple variable actuals may change. Lemma 5.3.6 recognizes side effects and if there are any, one must decide if the formal is evaluated more than once. (This requires only a minor modification to 5.3.2 and is left to the reader.) If it is repeatedly evaluated and there are side effects then 5.3.7 can be used to decide if it makes any difference in the result. This solves the problem for C_{vj}. q.e.d.

Theorem 5.4.2. For free schemata the membership problem for concidence classes C_{cr}, C_{cn} and C_{cj} is decidable.

Proof: From a syntactic check the duplicate actuals can be found. A modification of 5.3.2 will determine if the corresponding formals ever get assigned different values. If they do, lemma 5.3.7 determines if the differences affect the result. Membership in C_{cn} follows from 5.3.3, 5.3.6 and 5.3.7. For C_{cj} we need the arguments just given for the C_{cr} case, lemma 5.3.3 and the argument from the last theorem regarding side effects. q.e.d.

Theorem 5.4.3: For free schemata the membership problem for coincidence classes C_{rn}, C_{rj} and C_{nj} are decidable.

Proof: Membership in C_{rn} follows from 5.3.3, 5.3.6 and 5.3.7 as well as the

comments on side effects. For C_{rj} and C_{nj} the earlier comments on side effects apply together with lemma 5.3.3 for the former and lemmas 5.3.6 and 5.3.7 for the latter. q.e.d.

These results enable us to recognize that the facilities of a parameter evaluation mechanism are not fully utilized throughout the entire schema. Such a decision procedure, then, would enable a different "run time" implementation of parameter passing. Clearly, the usual case would be that some evaluations require one mechanism while others could get by on a "weaker" mechanism. If this information can be used to perform some optimizations, then the proofs of the above results are such that the analysis may be applied to these particular cases as well.

There can be no doubt that freedom is a very restrictive property. It has allowed us to recognize when the various mechanisms coincide. The fact that the property isn't decidable does not render our results useless since in compilers the *modus operandi* frequently assumes that the programs are free. If they are, then we know that a compiler can do a good job at translating from one parameter evaluation mechanism into another. If the programs are not free, then we do a less than perfect job, but we make no mistakes.

6. Summary.

To summarize the relationships discovered in the earlier chapters, we have,

$$P_{Rv} \equiv P_{Rc} \equiv P_{Rr} \equiv P_{Rn} \equiv P_{Rxg} \equiv P_{RxM} < P_{Rj} \equiv P_{Rjg} \leq P_{RjM} \equiv P_{Rq}$$

where x = v,c,r,n and the last two classes are universal. We showed that P_{Rj} cannot be effectively equivalent to P_{RjM}. We correlated our results with the notion of fixed point computation and our proofs suggest that although call by value, copy and reference do not, in general, compute the least fixed point, there is a constructive way to find schemata for which these mechanisms do compute the least fixed point. We were able to characterize all of our mechanisms in a simple and uniform way. Although the equivalence of two evaluation methods on a particular schema is generally undecidable, decision procedures were found for a restricted class. This class is consistent with the *modus operandi* of most compilers and for these, parameter evaluation optimizations could be realized.

Having considered several kinds of evaluation mechanisms, it is reasonable to ask how do the facilities compare. Certainly, the conventional wisdom has been called into question which says that call by name is "stronger" than call by value because it postpones evaluation. We would have to say that the important advantage of call by name over the weaker mechanisms is that it allows multiple evaluation of the formals with (potentially) different results each time. This facility is actually a very

restricted way of manipulating unevaluated functions as objects in the language. We saw in P_{Rq} that generalizing this facility achieved universality. If P_{Rj} were universal too, then it would be a far more attractive way (from an implementational standpoint) to achieve the computational power than a general function manipulating language.

Even though the weaker mechanisms are equivalent, a look at the proofs indicates that our study of classes of functionals, like other computability studies, does not tell the whole story. The efficiency degrades tremendously between P_{Rv} and P_{Rc} and between P_{Rr} and P_{Rn}. If our constructions cannot be made more optimal, then this suggests that there is a difference between these classes which we have not had the mechanism to recognize. Some complexity analysis might be useful in separating these, though its not quite clear how to formulate the questions. Alternatively, one could consider a stronger notion of equivalence, where all basic functions and predicates must be executed in the same order.

While we are on the subject of practicality, even if P_{Rj} turns out to be universal, a "practical man" would be less than excited with the necessarily nonconstructive nature of the equivalence. But pragmatically speaking, markers are generally available when performing all but the most bizarre programming exercises anyway, so P_{RjM} may be a more reasonable model of our use of call by name recursion.

We have left one open problem. The question of whether call by name is universal is closely related to open problems of [2] and [1]. (Notably, the question of whether program schemata augmented with two push-down stores with empty stack tests is universal.) Finding solutions to these, however, is probably best treated as recreational mathematics. We will learn very little new when we know the answer. Nor should much time be spent adding ones favorite language features to program schemata and comparing them to other classes. In the first place most common programming language constructs have already been examined and secondly, for n language features one can potentially form 2^n classes with each probably requiring a separate proof. Rather, future research in comparative schematology should be applied to finding out which "properties" determine the relationships which we've been studying. When we no longer have to produce a special proof for each new class we find, then we will have taken an important step forward.

7. Bibliography.

[1] Brown S., ***Program Schemata and Information Flow: A Study of Some Aspects of the Schema Power of Data Structures***, Cornell Univ. (Ph.D Thesis), CS Tech. Report TR 72 - 135 (1972)

[2] Brown, S., D. Gries and T. Szymanski, ***Program Schemata with Pushdown Stores***, Cornell Univ., CS Tech. Report TR 72 - 126 1972

[3] Cadiou, J. and Z. Manna, "Recursive Definitions of Partial Functions and Their Computations," Proc. of ACM Conf. on Proving Assertions About Programs, ACM, New york, 1972.

[4] Constable, R. L. and D. Gries, "On Classes of Program Schemata", SIAM J. Compt. 1, 1 (1972)

[5] Ekman, T. and C. Froberg, ***Introduction to*** ALGOL ***Programming***, Oxford University Press, London, 1967.

[6] Ershov, A., "Theory of Program Schemata," Proc. IFIP Cong. 1971.

[7] Fischer, M., "Lambda Calculus Schemata," Proc. of ACM Conf. on Proving Assertions About Programs, ACM, New York, 1972.

[8] Garland, S. and D. Luckham, "Translating Recursion Schemes into Program Schemes," Proc. of ACM Conf. on Proving Assertions About Programs, ACM, New York, 1972.

[9] Hewitt, C., ***More Comparative Schematology***, MIT Project MAC, Artificial Intelligence Memo No. 207 (1970)

[10] Hopcroft, J., and J. Ullman, ***Formal Languages and Their Relation to***

Automata, Addison - Wesley, Reading Mass. 1969.

[11] Ianov, I. "The Logical Schemes of Algorithms", 1958 (English translation in: *Problems of Cybernetics* 1, Pergamon Press, 1960)

[12] Itkin, V. and Z. Zwienogrodsky, "On Equivalence of Program Schemata", J. Comp. Syst. Sci. 1, (1972)

[13] Ledgard, H., "Ten Mini-Languages: A Study of Topical Issues in Programming Languages", Computer Surveys, 3, 3, (1971)

[14] Luckham, D. C., D. M. R. Park and M. S. Paterson, "On Formalized Computer Programs", J. Comp. Syst. Sci. 4, 3 (1970)

[15] Manna, A., S. Ness and J. Vuillemin, "Inductive Methods for Proving Properties of Programs," Proc. of ACM Conf. on Proving Assertions About Programs, ACM, New York, 1972.

[16] McCarthy, J., "A Basis for a Mathematical Theory of Computation", in *Computer Programming and Formal Systems*, P. Braffort and D Hirshberg, ed., North-Holland, Amsterdam, 1963

[17] Paterson, M. *Equivalence Problems in a Model of Computation*, (Ph.D Thesis), Trinity College, Cambridge, 1967 (available as MIT A.I. Technical Memo No. 1, (1970))

[18] Paterson, M. S. and C. E. Hewitt, "Comparative Schematology", Conf. Record of Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York, 1970

[19] Randell, B., and L. Russell, ALGOL 60 *Implementation*, Academic Press, New York, 1964

[20] Reynolds, J. "Definitional Interpreters for Higher Order Programming Languages," Tech. Rep., Sys. and Info. Sci. Dept., Syracuse University, 1972

[21] Rutledge, J. "On Ianov's Program Schemata," Jour. ACM 11, (1964)

[22] Strong, R., "Translating Recursion Equations into Flow Charts", J. Comp. Syst. Sci. 5, (1971)

[23] Weisenbaum, J. "The FUNARG Problem", MIT Mimeograph Report

Appendix I.

The following construction completes the details of Theorem 4.2.2 Note that the informal description given in the text over-simplified the problems. In particular, since the noncoercion facility of formal parameter assignment is used so extensively, all data manipulation must be done via formal parameters so that no subscript function can inadvertently become evaluated. To help the reader, the formals *s*, *t* and *temp* act as globals and are used strictly to transfer values. (Think of *s* and *t* as source and target, respectively, i.e. $t \leftarrow s$). In addition when a subscripted variable is to be fetched from, the formal *Afetch* is used to return to the earlier environment for the fetch and *Astore* is used analogously for stores. As was done in 3.3, no auxiliary function corresponding to a statement of the P_A schema returns until the computation is complete. Finally, most functions are used for manipulation of formals and hence their values do not matter. As a reminder to the reader, we always pass Ω as a **halt** argument in these cases and make assignments of such function's value to a variable *dummy*.

A remaining detail concerns the degenerate and erroneous cases (see 1.3). In particular, if a subscript value is used as though it were a domain value, the value should be taken as Ω. Conversly, if a domain value is used as a subscript, location zero is to be referenced. Note that it can be effectively decided *how* the value is to be used, although it cannot be decided what type it will be. We require, therefore, that two copies of all

variables and array locations must be maintained. One will be used to hold only valid values for subscripts and another valid values for domain elements. They are initialized to the function *zero* and the value Ω, respectively. Then the fetchs and stores are made with the proper element of each pair as dictated by the circumstances of the use. For identity assignments ($x \leftarrow y$) both terms are copied. *Note*, the construction given below, to avoid adding extra complexity, provides only a single cell for all variables. The reader can fill in these details allowing for ill-behaved schemata.

Construction: Let T be a schema in P_β. The following outline should provide sufficient detail to convince the reader that an equivalent schema in P_{Rq} can be effectively found.

(i) Label all instructions of the T schema and determine their successors as was done for the construction in 3.3. The constructions for (ii), (iii), (iv) below do not actually include all of the cases (e.g. $v_i \leftarrow v_j$), but we trust the reader can fill in the details.

(ii) Let <S> be a statement labeled L_i whose successor is L_j and envolving the reference of array values $A[v_i]$, $A[v_j]$, . . ., $A[v_k]$ and an assignment to $A[v_i]$. We define (schematically) an auxiliary function as follows.

$L_i(s,t,temp,Astore,Afetch,v_1,v_2, \ldots ,v_n)$:

temp ← v_i;

dummy ← Afetch;

Avi ← t;

. . . ;

temp ← v_k;

dummy ← Afetch;

Avk ← t;

<S'>;

s ← Avi;

temp ← v_j;

dummy ← Astore;

z ← L_j(s,t,Astore,Afetch,v_1,v_2, . . .,v_n);

halt(z)

where: v_1,v_2, . . .,v_n are simple variables in T

Avi, Avj, . . ., Avk are new identifiers called ***array referencing identifiers*** and are used in lieu of subscript expressions.

<S'> is the statement identical to <S> except with array referencing identifiers replacing the array references. The reader can make the obvious modifications in the event <S> is an **if** statement (and has two successors) or in the event it is a **halt** statement (and has no successors.)

(iii) Let L_i be an assignment of zero,

L_i: A[v_k] ← 0;

with successor statement L_j. Then form a new auxiliary function as

follows.

$L_i(s,t,temp,Astore,Afetch,v_1,v_2, \ldots,v_n)$:

temp ← v_k;

s ← zero(s,t,temp);

dummy ← Astore;

z ← $L_j(s,t,temp,Astore,Afetch,v_1,v_2, \ldots,v_n)$;

halt(z)

(iv) Let L_i be an increment statement, i.e.

L_i: $v_p \leftarrow v_q + 1$;

with successor statement L_j. Then form a new auxiliary function following the schema:

$L_i(s,t,temp,Astore,Afetch,v_1,v_2, \ldots,v_n)$:

local ← Ω;

dummy ← increment(s,t,temp,v_q);

v_p ← temp;

z ← L_j(s,t,temp,storechain(s,t,temp,local,Astore),

fetchchain(s,t,temp,local,Afetch),$v_1,v_2, \ldots,v_n$);

halt(z)

(v) Define the following <rec function> and auxiliary functions.

$(u_1,u_2, \ldots ,u_t)$:

source ← Ω;

target ← Ω;

temporary ← Ω;

z ← firstcall(source,target,temporary);

halt(z)

firstcall(s,t,temp):

local ← Ω;

z ← L_1(s,t,temp,storend(s,t,temp,local),fetchend(s,t,temp,local),

$v_1,v_2, \ldots ,v_n$);

halt(z)

fetchend(s,t,temp,local):

s ← local;

dummy ← temp;

halt(Ω)

storend(s,t,temp,local):

t ← local;

dummy ← temp;

local ← t;

halt(Ω)

fetchchain(s,t,temp,local,pred):

temp ← pred;

s ← local;

dummy ← temp;

halt(Ω)

storechain(s,t,temp,local,pred):

temp ← pred;

t ← local;

dummy ← temp;

local ← t;

halt(Ω)

zero(s,t,temp): t ← s; temp ← Ω; **halt**(Ω)

increment(s,t,temp,chain):

temp ← index1(s,t,temp,chain);

halt(Ω)

index1(s,t,temp,chain): temp ← chain; **halt**(Ω)

where: $u_1,u_2, \ldots,u_t$ are the input variables to T.

$v_1,v_2, \ldots,v_n$ are the simple variables of T.

Appendix II.

The following construction completes the details of Theorem 4.2.3.

Let Tm be a Turing machine over two input symbols {0,1} with a tape one-way infinite to the right. Let Q be the discription quintuples,

<state,symbol,newstate,newsymbol,shift> ∈ Q.

We will employ a technique which differs little from that used in Appendix I. The only subtlety is in the fact that for that simulation the indefinite recursion was done explicitly with each procedure calling its successor(s), while for this simulation we must perform the indefinite recursion by executing a formal (*ex*) so that the decision to **halt** can be made. The tape will be created, one cell at a time with local variables. (Whereas in Appendix I the local was the storage cell of the defining environment for that cell, this construction must pass the cell as the storage for the next calling environment. (This is because the tape must be initialized to blank where before it was initialized to Ω.) The tape symbols will be functions, ***blank*** and ***mark***. These functions will manipulate global formals (***zero*** and ***one***, respectively) to decide which is the next state. Writing on the tape and movement of the read head are done analogously to that for array assignment and index incrementing. The remainder of the details should be clear.

Construction. The following steps will produce a schema $T \in P_{Rq}$ with the requirements that if T halts then Tm halts on blank tapes.

(i) Let <i,x,j,symbol,shift> be a state quintuple, where *i* is the state number, *x* the symbol read, *j* the new state, *symbol* the symbol written and *shift* the direction the tape read head is to move. Form an auxilary function *ix*state according to the following schema, such that:

(a) 1* or 2* is chosen depending on whether the written symbol is 0 or 1, respectively.

(b) 3* or 4* is chosen depending upon whether the head is to be shifted right or left, respectively.

```
ixstate(temp,ex,zero,one,read,write,head,symbol,cell):
    local ← Ω;
    cell ← blank(ex,zero);
1*  symbol ← blank(ex,zero);
2*  symbol ← mark(ex,one);
    temp ← head;
    dummy ← write;
3*  head ← shift(temp,ex,symbol,head);
4*  begin
        temp ← head;
        dummy ← temp;
        head ← temp   end
    zero ← j0state(temp,ex,zero,one,taperead(temp,ex,zero,one,cell,
                symbol,read), tapewrite(temp,ex,cell,symbol,write),
```

head,symbol,local);

one ← *j*1state(temp,ex,zero,one,taperead(temp,ex,zero,one,cell,

symbol,read),tapewrite(temp,ex,cell,symbol,write),

head,symbol,local);

temp ← head;

dummy ← read;

dummy ← ex;

halt(Ω)

Note that the transition is realized by loading *zero* and *one* with the proper successor cells and then invoking *read* which executes the tape cell function. The tape cell function then transfers the next state function description into *ex* which is executed in the assignment to *dummy*

(ii) For all halt pairs <i,x> generate: *ix*state(a,b,c,d,e,f,g,h,i): **halt**(Ω)

(iii) Generate the following utility functions:

firstcall(temp,ex,zero,one,head,symbol,cell):

local ← Ω;

cell ← blank(ex,zero);

head ← shift(temp,ex,symbol,pass(temp,ex,symbol));

dummy ← 10state(temp,ex,zero,one,readend(temp,ex,zero,one,cell,symbol),

writeend(temp,ex,cell,symbol),head,symbol,local);

halt(Ω)

```
pass(temp,ex,symbol): ex ← symbol; temp ← Ω; halt(Ω)


readend(temp,ex,zero,one,cell,symbol):
    ex ← null;
    symbol ← cell;
    dummy ← temp;
    dummy ← ex;
    halt(Ω)


writeend(temp,ex,cell,symbol):
    ex ← cell;
    dummy ← temp;
    cell ← ex;
    halt(Ω)


taperead(temp,ex,zero,one,cell,symbol,pred):
    dummy ← pred;
    ex ← null;
    symbol ← cell;
    dummy ← temp;
    dummy ← ex;
    halt(Ω)
```

```
tapewrite(temp,ex,cell,symbol,pred):
    dummy ← pred;
    ex ← cell;
    dummy ← temp;
    cell ← ex;
    halt(Ω)

null: halt(Ω)

shift(temp,ex,symbol,chain): temp ← chain; halt(Ω)
```

(iv) Generate the following <rec program> and augment it with the auxiliary functions from (i) - (iii):

```
(x): temporary ← Ω;
    local ← Ω;
    execute ← Ω;
    zerochoice ← Ω;
    onechoice ← Ω;
    tapehead ← Ω;
    writingsymbol ← Ω;
    dummy ← firstcall(temporary,execute,zerochoice,onechoice,tapehead,
                writingsymbol,local);
    halt(Ω)
```

Note that the input variable is unused and is provided only to meet syntactic requirements.